

Un lenguaje de dominio específico para mutación de modelos



Pablo Gómez-Abajo

Directores: Esther Guerra Sánchez, Mercedes García Merayo

Departamento de Ingeniería Informática
Universidad Autónoma de Madrid

Se presenta esta memoria para la obtención del título de
Doctorado en Ingeniería Informática y de Telecomunicaciones

Dedico esta tesis a Almudena.

Los sueños, sueños son. Aunque a veces los sueños se hacen realidad.

Esto suele ocurrir cuando uno menos se lo espera.

Declaración

Declaro que excepto las referencias específicas hechas al trabajo de otros, los contenidos de esta memoria son originales y no se han enviado en conjunto o en parte para consideración en ningún otro título o cualificación, ni en ninguna otra universidad. Esta memoria es mi propio trabajo y no contiene nada más que el resultado del trabajo realizado en colaboración con otros, excepto lo especificado en el texto.

Pablo Gómez-Abajo
Abril 2020

Agradecimientos

En primer lugar, quiero agradecer de forma infinita a Esther Guerra, Juan de Lara y Mercedes G. Merayo su trabajo de orientación y tutorización para que esta tesis haya llegado a buen puerto, tan infinita como la paciencia que han tenido conmigo. El 99,9 % de las veces ellos veían más lejos que yo en los aspectos relevantes del trabajo, y el asombro es tal que sólo puedo quitarme el sombrero. El 0,1 % restante consistió básicamente en conseguir hacer lo que me decían e ir limando poco a poco mi cabezonería. Se puede decir que mi trabajo en estos años ha consistido básicamente en seguir los pasos que me iban marcando. De ellos quiero destacar aquí no sólo su incuestionable calidad como investigadores, sino por encima de todo su inestimable calidad humana.

En segundo lugar, quiero agradecer a Almudena, mi compañera en el camino de la vida. Ella es licenciada en Teología, maestra de religión en la enseñanza secundaria, y en mi opinión, su disciplina, que se puede considerar como una “Ingeniería de almas”, supone una complejidad y un valor mucho más elevados que los de cualquiera de las ingenierías habituales. Es gracias a ella que me animé a iniciar mi carrera como investigador. A ella le debo en definitiva todo esto. Es también un asombro comprobar que cuando se comparten los sueños todo resulta mucho más sencillo. Te quiero.

En tercer lugar, quiero agradecer también a Víctor Moreno Serna y a Juan Vázquez, los únicos amigos que realmente conservo de la adolescencia. Creo que ambos ocupan un lugar de privilegio en mi vida ya desde hace bastantes años. Todo esto hubiese resultado mucho más aburrido, y tengo serias dudas de haber podido con ello, de no ser por nuestras periódicas reuniones para ver al Rayo o tomar algo en la mañana de los sábados. Gracias también a toda la gente del Rayo, esto también va por Vallecas.

En cuarto lugar, agradezco también a Francisco Gil y a César Sánchez, otros dos buenos amigos con los que he podido contar siempre que les he necesitado.

Quiero mostrar mis agradecimientos también a la empresa Ingeniería y Prevención de Riesgos (I+P) por los años que estuve trabajando con ellos. Creo que esa época ha resultado fundamental en mi formación como ciudadano. Después de unos años me di cuenta de que había aprendido mucho acerca de trasladar sus ideas a soluciones informáticas, pero que también empleaba mucho tiempo repitiendo gran parte del trabajo que hacía. Es por esto

que quedé cautivado por las técnicas de modelado que pacientemente me explicaban Juan y Esther en las primeras entrevistas para incorporarme al grupo Miso. Considero además que trabajar en I+D fue también un entrenamiento excelente para soportar la presión del trabajo durante la tesis.

Agradezco también a mis padres y a mis dos hermanos. Durante muchos años han sido mi soporte fundamental, siempre presentes en las buenas y en las malas, y les estoy inmensamente agradecido, como inmensa es también mi alegría de que todos estemos saliendo adelante, cada uno aportando su granito de arena. Quiero destacar aquí en especial el papel de mi padre y la influencia tan positiva que siempre ha tenido sobre mí. Él es Ingeniero de Caminos y ha ejercido esta profesión durante gran parte de su vida, con un sacrificio y una precisión tales que creo que es bien difícil que yo le pueda alcanzar algún día. Se puede decir que mientras mi padre hacía puentes, yo sólo los estoy cruzando. También quiero agradecer especialmente a mi tío Quique, al que le debo en definitiva haber descubierto mi vocación por la informática. Agradezco también a todos mis demás familiares, en especial a mi abuela Sole, a mi tía Seve, y a mi tía Mercedes, que fallecieron en el período en el que he estado realizando la tesis. Sin duda, todos esos buenos recuerdos compartidos de mi infancia y adolescencia perdurarán por siempre. También incluyo aquí mis agradecimientos a la familia de Almudena, que me acogió como un miembro más desde el primer día.

Agradezco a todos los compañeros que he tenido en el grupo Miso durante estos años, en especial a Antonio Garmendía. He aprendido mucho de todos ellos y considero que sin su compañía, buen humor, y ayuda, todo este trabajo se hubiese hecho mucho más cuesta arriba. Mis agradecimientos van también para mis amigos en la carrera y en el máster, en especial para Mickael Guisado Boucard y Raquel Díaz López, creo que éramos los tres únicos bichos raros en la carrera. Creo sin duda que si he conseguido este objetivo es en gran medida gracias a ellos.

También quiero agradecer aquí a todos los profesores que he tenido, especialmente a Manuel Alfonseca Moreno y a Josefina Sierra Santibáñez. Creo que es de justicia reconocer que todos ellos tienen mucho que ver en que finalmente yo haya podido superar estos estudios tan exigentes y con tan buenos resultados. A todos ellos les deseo que las cosas les vayan bien siempre. Aunque muchas veces no me he portado demasiado bien como estudiante, siempre he considerado que si hay algo en lo que España destaca de manera excepcional es en la calidad de los docentes que tenemos.

Por último, quiero agradecer también a todas las demás personas que se han tropezado conmigo por este camino apasionante. Creo que he intentado aprender de todos ellos, y estoy seguro de que a cada uno de ellos les corresponde una parte del hombre que soy hoy, por pequeña que sea.

Sin más, me da la impresión de que la etapa más difícil de mi vida ya está superada, y espero vivir el tiempo suficiente para recoger muchos de sus frutos. Aunque puede que también esté equivocado, y esto no haya hecho más que empezar.

Resumen

Las técnicas de mutación de software se utilizan en campos diversos como las pruebas de mutación, la pruebas de programas, la prueba de fórmulas lógicas, los algoritmos genéticos y la generación automática de ejercicios. Las soluciones existentes suelen utilizar un enfoque a medida, construyendo desde cero una solución específica para el problema que pretenden abordar. Este enfoque conlleva un esfuerzo elevado de implementación para cada caso, es propenso a errores, supone un alto coste de mantenimiento, y tiene como resultado la creación de numerosas soluciones cada una de las cuáles sólo resuelve un problema en concreto.

Con el objetivo de proporcionar un enfoque genérico que ayude a superar estos inconvenientes, esta tesis presenta un lenguaje de dominio específico para mutación de modelos llamado `WODEL`, y su entorno de desarrollo. `WODEL` es independiente del dominio y puede utilizarse con cualquier lenguaje definido por medio de un meta-modelo. Incluye primitivas de mutación para creación, modificación, borrado, retipado y clonado de objetos, y para creación, modificación y borrado de referencias. `WODEL` proporciona facilidades de soporte al proceso de mutación, como la validación de los mutantes, un registro de las mutaciones aplicadas y la detección de mutantes equivalentes. También da soporte a la ingeniería de operadores de mutación mediante la generación de métricas de mutación y la síntesis automática de modelos semilla que aseguran la cobertura de todas las instrucciones de un programa `WODEL`, facilitando de este modo su prueba y validación.

Además, `WODEL` es extensible y permite aprovechar sus funcionalidades para la mutación de modelos en aplicaciones de post-procesado. Para ilustrar dicha extensibilidad, esta tesis presenta dos de estas extensiones a `WODEL`: una para la generación automática de ejercicios de auto-evaluación para estudiantes, a la que se ha llamado `WODEL-EDU`; y otra para facilitar la creación de herramientas de pruebas de mutación para lenguajes de programación o de modelado, a la que se ha llamado `WODEL-TEST`.

Abstract

Software mutation techniques are used in different fields such as mutation testing, software testing, logic formulas testing, genetic algorithms and the automated generation of exercises. The existing solutions are usually ad-hoc, creating from scratch a specific solution for the faced problem. This approach involves a high implementation effort, it is error-prone, entails a high maintenance cost, and results in a variety of solutions each of which only solves a particular problem.

With the purpose of providing a generic approach that alleviates these inconveniences, this thesis introduces a domain-specific language for model mutation called `WODEL`, and its development environment. `WODEL` is domain independent and can be used with any arbitrary language defined by a meta-model. It includes mutation primitives to create, modify, delete, retype and clone objects, and to create, modify and delete references. `WODEL` provides facilities to simplify the mutation process, such as model validation, a registry of the applied mutations and the detection of equivalent mutants. It also supports the engineering of mutation operators by the generation of metrics of the mutation programs and the automated synthesis of seed models which ensure full coverage of the statements in a `WODEL` program, hence easing its testing and validation.

Additionally, `WODEL` is extensible and permits taking advantage of all its model mutation functionalities in post-processing applications. To illustrate this extensibility capability, this thesis reports on two of these extensions to `WODEL`: the first one targeted to the automated generation of self-assessment exercises for students, called `WODEL-EDU`; and the second one targeted to ease the creation of mutation testing tools for programming or modelling languages, called `WODEL-TEST`.

Índice general

Declaración	V
Agradecimientos	VII
Resumen	XI
Abstract	XIII
Índice de figuras	XIX
Índice de tablas	XXI
Nomenclatura	XXIII
1. Introducción	1
1.1. Antecedentes	1
1.2. Motivación	2
1.3. Contribución	3
1.3.1. Contribución técnica	4
1.3.2. Publicaciones	5
1.4. Financiación	7
1.5. Estructura de la memoria	7
2. Conceptos Previos y Trabajo Relacionado	9
2.1. Conceptos previos sobre MDE	9
2.1.1. Ingeniería dirigida por modelos	9
2.1.2. Lenguajes de dominio específico	13
2.1.3. Transformación de modelos	14
2.1.4. Generación de texto	17
2.1.5. Tecnologías utilizadas	17

2.1.5.1.	Eclipse Modeling Framework	18
2.1.5.2.	Xtext y Xtend	19
2.2.	Técnicas de mutación	20
2.2.1.	Aplicaciones de las técnicas de mutación	20
2.2.2.	Pruebas de mutación	22
2.2.3.	Lenguajes de definición de operadores de mutación	28
2.3.	Generación automática de ejercicios	30
2.4.	Conclusiones del estudio	33
3.	Wodel: Un DSL para Mutación de Modelos	35
3.1.	El DSL WODEL	35
3.2.	Soporte al proceso de mutación	46
3.2.1.	Validación de mutantes	47
3.2.2.	Registro de mutaciones	47
3.2.3.	Detección de mutantes equivalentes	49
3.2.4.	Post-procesado de mutantes	50
3.3.	Soporte a la definición de programas de mutación	50
3.3.1.	Síntesis de modelos semilla	50
3.3.2.	Métricas de mutación	55
3.3.2.1.	Métricas estáticas	55
3.3.2.2.	Métricas dinámicas	58
3.4.	Herramienta proporcionada	58
3.5.	Evaluación de la concisión de Wodel	65
4.	Wodel-Edu: Generación Automática de Ejercicios	69
4.1.	Arquitectura	70
4.2.	Lenguajes de configuración de ejercicios	72
4.2.1.	Describiendo los ejercicios con EDUTEST	72
4.2.2.	Describiendo la visualización de los modelos con MODELDRAW	73
4.2.3.	Descripción textual de los elementos del modelo con MODELTEXT	74
4.2.4.	Descripción textual de los operadores de mutación con MUTATEXT	75
4.3.	Generación de ejercicios	76
4.4.	Evaluación de la calidad de los ejercicios generados	80
5.	Wodel-Test: Pruebas de Mutación Independientes del Lenguaje	85
5.1.	Visión general	85
5.2.	Herramienta proporcionada	88

5.3. Evaluación	92
5.3.1. Comparativa de la herramienta de pruebas de mutación generada para Java con herramientas hechas manualmente	94
5.3.1.1. Definición de la herramienta de pruebas de mutación para Java	95
5.3.1.2. Características generales	96
5.3.1.3. Extensibilidad de los operadores de mutación	98
5.3.1.4. Eficiencia y eficacia	99
5.3.1.5. Discusión de los resultados y aspectos que pueden afectar a la validez	100
5.3.2. Caso de estudio: Construyendo una herramienta de pruebas de mutación para ATL	101
5.3.2.1. El lenguaje ATL	102
5.3.2.2. Utilizando WODEL-TEST para construir una herramienta de pruebas de mutación para ATL	104
5.3.2.3. Evaluando el proceso de creación de la herramienta de pruebas de mutación para ATL	108
5.3.2.4. Discusión de los resultados y aspectos que pueden afectar a la validez	110
6. Conclusiones y Trabajo Futuro	113
6.1. Conclusiones	113
6.2. Trabajo futuro	114
Bibliografía	117
Anexo A. Gramática de WODEL	127
Anexo B. Gramática de EDUTEST	131
Anexo C. Gramática de MODELDRAW	133
Anexo D. Gramática de MODELTEXT	135
Anexo E. Gramática de MUTATEXT	137
Anexo F. Operadores de Mutación para Autómatas Finitos	139
Anexo G. Operadores de Mutación para el Meta-modelo de Java de MODISCO	141

Anexo H. Operadores de Mutación para ATL

149

Índice de figuras

1.1. Mutación de modelos de un autómata finito	2
2.1. Pirámide de 4 niveles de MOF.	11
2.2. Meta-modelo para autómatas finitos.	12
2.3. Modelo de autómata finito que acepta los números binarios pares.	13
2.4. Transformación de modelos <i>out-place</i>	15
2.5. Mutación de modelos.	17
2.6. Arquitectura de un generador de código.	18
2.7. Proceso de pruebas de mutación.	26
3.1. Algunos operadores de mutación soportados.	36
3.2. Ejemplo de operador de mutación de creación de objetos.	37
3.3. Ejemplo de operador de mutación de clonación de objetos.	38
3.4. Ejemplo de operador de mutación de modificación de referencia.	39
3.5. Ejemplo de operador de mutación de modificación de atributo.	39
3.6. Ejemplo de operador de mutación de modificación de destino de referencia.	39
3.7. Ejemplo de operador de mutación de retipado de objetos.	40
3.8. Ejemplo de operador de mutación de borrado de objetos.	40
3.9. Estrategias de selección soportadas.	42
3.10. Meta-modelo del registro de los operadores de mutación (fragmento).	48
3.11. Generación de modelos semilla para un programa <code>WODEL</code>	51
3.12. Modelo semilla generado.	54
3.13. Arquitectura del entorno de desarrollo de <code>WODEL</code>	59
3.14. Página de preferencias de <code>WODEL</code>	61
3.15. Asistente de <code>WODEL</code> para la generación de modelos semilla.	62
3.16. El entorno <code>WODEL</code> en acción.	63
3.17. Meta-modelo con la métrica estática.	64
4.1. Esquemas de generación de los tres tipos de ejercicios.	70

4.2.	Arquitectura de WODEL-EDU.	71
4.3.	Captura de pantalla de un ejercicio de <i>respuesta alternativa</i>	78
4.4.	Captura de pantalla de un ejercicio de <i>elección de diagrama múltiple</i>	78
4.5.	Captura de pantalla de un ejercicio de <i>selección de reparación múltiple</i>	79
4.6.	Pasos para generar el ejercicio en la Figura 4.5.	80
(a).	Autómata semilla.	80
(b).	Primer autómata mutado utilizado para producir reparaciones correctas, y mostrado en el ejercicio.	80
(c).	Segundo autómata mutado utilizado para producir reparaciones incorrectas.	80
4.7.	Puntuación media de las páginas de ejercicios consideradas en la evaluación.	83
4.8.	Nota media de los participantes en las páginas de ejercicios.	84
5.1.	Creación y utilización de una herramienta de pruebas de mutación con WODEL-TEST.	86
5.2.	Esquema del funcionamiento de la herramienta de pruebas de mutación para Java.	88
5.3.	Arquitectura de WODEL-TEST.	89
5.4.	Selección de operadores de mutación en la ventana de preferencias (extracto).	90
5.5.	Vista global de los resultados de pruebas de mutación.	91
5.6.	Vistas detalladas de los resultados del proceso de pruebas de mutación.	93
5.7.	Vista con los mutantes generados por cada operador de mutación.	94
5.8.	Subconjunto del meta-modelo de Java (obtenido de MoDisco [26]).	95
5.9.	Esquema de transformaciones modelo a modelo con ATL.	102
5.10.	Ejemplo de transformación ATL.	103
5.11.	Extracto del meta-modelo de ATL.	105
5.12.	Entorno de pruebas de mutación para ATL creado con WODEL-TEST.	107

Índice de tablas

2.1. Resumen de los trabajos de mutación analizados (a).	23
2.2. Resumen de los trabajos de mutación analizados (b).	24
2.3. Resumen de los trabajos de mutación analizados (c).	25
3.1. Plantillas para generar las restricciones OCL a partir de las primitivas de mutación.	52
3.2. Métricas estáticas del meta-modelo para el operador de mutación definido en el Listado 3.9. C=Creación, M=Modificación, D=Borrado, (a)=atributo, (r)=referencia.	57
3.3. Reglas para calcular las acciones implícitas a las que da lugar cada acción explícita.	57
3.4. Líneas de código Java requeridas para implementar cada primitiva de mutación WODEL.	67
5.1. Comparativa de herramientas de pruebas de mutación para Java.	97
5.2. Aplicación de las herramientas de pruebas de mutación para Java sobre el proyecto functional-matrix-operator.	99
5.3. Operadores de mutación para ATL.	106
5.4. Comparativa de herramientas de pruebas de mutación para ATL.	111
F.1. Operadores de mutación para Autómatas Finitos utilizando WODEL.	140
G.1. Operadores de mutación para el meta-modelo de Java de MoDisco utilizando WODEL.	147
H.1. Operadores de mutación para ATL utilizando WODEL.	150

Nomenclatura

Acrónimos / Abreviaturas

ATL Atlas Transformation Language

DSL Domain-Specific Language - Lenguaje de Dominio Específico

LOC Lines Of Code - Líneas de código

MDE Model-Driven Engineering - Ingeniería Dirigida por Modelos

MT Mutation Testing - Pruebas de mutación

Capítulo 1

Introducción

En este capítulo se introducen la motivación y los objetivos de esta tesis. En la Sección 1.1 se describen los antecedentes de este trabajo. A continuación, en la Sección 1.2 se explica la motivación identificada para realizarlo. En la Sección 1.3 se indican las contribuciones que ha supuesto. En el apartado siguiente, la Sección 1.4 detalla la financiación con la que se ha contado para su desarrollo. Por último, la Sección 1.5 describe la estructura de este documento.

1.1. Antecedentes

La Ingeniería Dirigida por Modelos (en inglés *Model-Driven Engineering*, MDE) [24] utiliza los modelos en todas las etapas del proceso de desarrollo de software para especificar, simular, analizar, verificar y generar código para el sistema final, entre otras actividades. Estos modelos se definen normalmente utilizando lenguajes de dominio específico (en inglés *Domain-Specific Languages*, DSLs [132]) especializados en el dominio de la aplicación en particular. Los dominios en los que MDE y los DSLs se han aplicado con éxito incluyen la programación concurrente [83], la programación de procesos reactivos de control [100], el modelado de procesos de negocio [22], el modelado web [55, 28], y el desarrollo de software para tarjetas inteligentes [106], por mencionar algunos.

Dado que los modelos son el principal activo en MDE, la manipulación de modelos se convierte en una actividad clave en este paradigma. Con este propósito, DSLs orientados específicamente a la tarea de transformación de modelos se utilizan en gran medida. Entre otros, existen DSLs para especificar simuladores de modelos [123, 114], para producir un modelo a partir de otro [65], para migrar modelos [103], o para sintetizar código [85].

La mutación de modelos es un tipo de transformación de modelos que genera un conjunto de variantes (o modelos *mutantes*) a partir de uno o más modelos *semilla* de partida, por

medio de la aplicación de uno o más *operadores de mutación*. La mutación de modelos tiene muchas aplicaciones. Por ejemplo, en las pruebas de transformación de modelos [10], una transformación se representa como un modelo que se muta para evaluar la eficacia del conjunto de pruebas de la transformación. Este conjunto de pruebas se puede crear mediante la mutación de un conjunto de modelos semilla de entrada. En el ámbito educativo, se puede representar un problema y su solución mediante un modelo, y mutarlo para generar ejercicios de auto-evaluación donde el estudiante debe detectar los errores introducidos [50, 104].

En la Figura 1.1 se muestra la aplicación de mutación de modelos al dominio de los autómatas finitos. El modelo semilla se corresponde con un autómata finito que acepta el lenguaje definido por los números binarios pares. En la parte derecha de la figura se muestran dos ejemplos de modelos mutantes generados a partir del modelo semilla tras aplicar el operador de mutación de intercambiar el símbolo de dos transiciones. Como se puede observar, estos dos autómatas finitos mutantes generados aceptan lenguajes diferentes al aceptado por el autómata finito original. Estos mutantes podrían utilizarse con distintos propósitos, como por ejemplo, la generación de ejercicios [50].

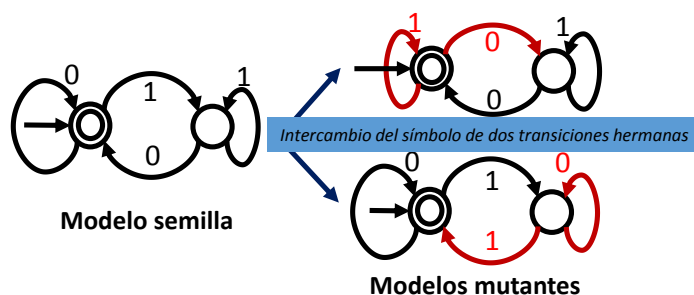


Figura 1.1 Mutación de modelos de un autómata finito.

1.2. Motivación

Existen tres enfoques principales para implementar operadores de mutación: utilización de un lenguaje de programación de propósito general [82]; utilización de un lenguaje de transformación [10]; o utilización de un framework específico de mutación [30]. Los lenguajes de programación de propósito general y los lenguajes de transformación no están orientados a la definición de operadores de mutación y generación de mutantes. Por un lado, estos enfoques no proporcionan primitivas de primer nivel para configurar el número de mutantes que se quiere generar, que puede ser un número determinado, un rango, o de modo exhaustivo, es decir, un mutante para cada posible localización en la que el operador de mutación puede aplicarse. Tampoco garantizan que los mutantes generados sean conformes al meta-modelo

del dominio o que cumplan alguna restricción adicional que pueda ser de interés para el problema tratado. Por ejemplo, el proceso de mutación de la Figura 1.1 debería asegurar que los mutantes generados son autómatas válidos. Por último, los lenguajes de programación y de transformación de propósito general no proporcionan funcionalidades específicas de un proceso de mutación, como puede ser la detección de mutantes duplicados. En cuanto a los frameworks para mutación de modelos existentes, éstos son específicos para un lenguaje (por ejemplo, fórmulas lógicas [60]) o para un dominio (por ejemplo, pruebas [10, 16]). Por tanto, detectamos que existe una falta de propuestas que faciliten la definición de operadores de mutación mediante un lenguaje de alto nivel apropiado para esta tarea y que se puedan aplicar a lenguajes o aplicaciones arbitrarios. De este modo, se facilitaría la creación de entornos de mutación avanzados para nuevos dominios con funcionalidades asociadas con este proceso tales como soporte a la trazabilidad de mutaciones aplicadas, métricas de mutación, o detección de mutantes equivalentes. Por otra parte, la generación de mutantes tiene como propósito su uso en procesos posteriores, que habitualmente deben realizarse manualmente. Actualmente no existen herramientas para facilitar la integración de aplicaciones externas que utilicen los mutantes generados.

1.3. Contribución

Para facilitar la especificación de operadores de mutación y creación de modelos mutantes para cualquier dominio de aplicación arbitrario, proponemos un DSL al que hemos llamado `WODEL`. Este lenguaje proporciona primitivas para mutación de modelos como, por ejemplo, creación, borrado, retipado (es decir, cambio de tipo del objeto), clonado de objetos, redirección de referencias y modificación de atributos. También permite utilizar diversas estrategias de selección de elementos para la aplicación de las mutaciones como selección aleatoria, específica, o basada en alguna propiedad del elemento. `WODEL` también permite crear operadores de mutación compuestos de una secuencia de operadores simples.

La ejecución de un programa `WODEL` asegura que los modelos mutantes generados son válidos, es decir, son conformes al meta-modelo (ver Capítulo 2) y satisfacen las restricciones especificadas. También garantiza la asignación automática de valores apropiados a los atributos y referencias obligatorios que no se hayan inicializado en el programa, y la selección automática del contenedor para los objetos creados si no se ha indicado ninguno de forma explícita. Otras características proporcionadas por `WODEL` en cuanto a la ejecución del programa son la identificación de mutantes duplicados, y el registro de las mutaciones aplicadas.

El entorno de desarrollo construido permite crear programas `WODEL` y su compilación a Java, y puede extenderse con opciones de post-procesado que requieran otras aplicaciones. Como soporte al desarrollo de programas de mutación, el entorno de desarrollo que hemos construido para `WODEL` proporciona una serie de métricas estáticas y dinámicas que proporcionan información sobre la cobertura del programa `WODEL`. Estas métricas indican el número de operaciones de mutación de cada tipo (creación, borrado y modificación) que se incluyen en el programa `WODEL`, y las clases del meta-modelo sobre las que estas instrucciones se aplican. Se distingue entre métricas estáticas que se calculan a partir del programa `WODEL`, o dinámicas que dan información detallada de una ejecución concreta del mismo. Por último, el entorno permite la generación automática de modelos semilla sobre los que todas las instrucciones incluidas en el programa tienen aplicación.

En esta memoria se describen las dos extensiones de `WODEL` que hemos desarrollado: la extensión para la generación automática de ejercicios, a la que hemos llamado `WODEL-EDU`; y la extensión para la generación de entornos de pruebas de mutación para lenguajes de modelado o programación, a la que hemos llamado `WODEL-TEST`. La generación de ejercicios mediante `WODEL-EDU` se realiza mutando los modelos con las soluciones. Las pruebas de mutación se utilizan principalmente para evaluar la calidad de las pruebas ya existentes. Ésta técnica consiste en introducir artificialmente cambios en el código de los programas, y medir cuántos de esos cambios son detectados por los conjuntos de pruebas. Crear entornos de pruebas de mutación para un lenguaje puede ser costoso, y por eso `WODEL-TEST` resulta de utilidad. Además, se han implementado dos entornos de pruebas de mutación utilizando la extensión `WODEL-TEST`: uno para el lenguaje Java, y otro para ATL, que resulta especialmente novedoso ya que es el primero que se propone.

Las contribuciones de esta tesis ha sido evaluadas del siguiente modo. Por un lado, hemos evaluado la concisión de `WODEL` para expresar programas de mutación, mediante su comparación con la implementación equivalente en el lenguaje de programación Java. Además, se ha realizado una evaluación con usuarios de la herramienta `WODEL-EDU` de generación automática de ejercicios midiendo la calidad de los ejercicios generados. Por último, se ha evaluado `WODEL-TEST` mediante la creación de dos entornos de pruebas de mutación, uno para Java y otro para el lenguaje de transformación de modelos ATL [12]. Además, con objeto de comprobar hasta qué punto los entornos generados con nuestro enfoque son realistas, los hemos comparado con otros entornos existentes de pruebas de mutación para Java y ATL, obteniendo muy buenos resultados.

1.3.1. Contribución técnica

Esta memoria proporciona las siguientes contribuciones técnicas:

1. Diseño e implementación del DSL `WODEL`, que incluye:
 - Primitivas de mutación de modelos y estrategias de selección de objetos.
 - Entorno de desarrollo integrado en Eclipse con completado de código y comprobación de tipos.
 - Métricas estáticas y dinámicas de programas `WODEL`, referentes a su cobertura del meta-modelo y de los operadores de mutación.
 - Síntesis de modelos semilla a partir del programa `WODEL`.
 - Punto de extensión para registrar nuevas opciones de post-procesado de mutantes, así como criterios de detección de mutantes duplicados y equivalentes.
2. Diseño e implementación de la herramienta `WODEL-EDU` para generación automática de ejercicios, que incluye:
 - Lenguajes de configuración de los ejercicios: el DSL `EDUTEST` para la descripción del estilo de los ejercicios; el DSL `MODELDRAW` para representar un modelo gráficamente; el DSL `MODELTEXT` para representar un modelo en texto; y el DSL `MUTATEX` para representar los operadores de mutación aplicados en texto.
 - Soporte para tres tipos de ejercicios: respuesta alternativa; selección de un diagrama entre varios; y selección de correcciones al diagrama.
 - Se proporciona un ejemplo de aplicación de `WODEL-EDU` para la generación automática de ejercicios de autómatas finitos.
3. Diseño e implementación de la herramienta `WODEL-TEST` para la creación de entornos de pruebas de mutación para lenguajes de programación o modelado, que incluye:
 - Soporte para la configuración del entorno de pruebas de mutación para un lenguaje dado.
 - Soporte para la generación de métricas de mutación de un conjunto de pruebas en el lenguaje configurado.
 - Se proporcionan dos casos de aplicación de `WODEL-TEST` para la creación de entornos de pruebas de mutación para los lenguajes Java y ATL.

1.3.2. Publicaciones

La presente tesis ha supuesto las siguientes publicaciones:

Revistas (2):

1. P. Gómez-Abajo, E. Guerra, y J. de Lara. A domain-specific language for model mutation and its application to the automated generation of exercises. *Computer Languages, Systems & Structures*, 49:152 – 173, 2017. Elsevier. Índice de impacto: JCR 2017: 1,840. Q2 en Computer Science / Software Engineering.
2. P. Gómez-Abajo, E. Guerra, J. de Lara, y M. G. Merayo. A tool for domain independent model mutation. *Science of Computer Programming*, 163:85–92, 2018. Elsevier. Índice de impacto: JCR 2018: 1,088 Q3 en Computer Science / Software Engineering.

Revistas en proceso de revisión (1):

1. P. Gómez-Abajo, E. Guerra, J. de Lara, y M. G. Merayo. A model-based framework for language-independent mutation testing. *Software and Systems Modeling*. Springer. Índice de impacto: JCR 2018: 2,660. Q1 en Computer Science / Software Engineering. En segunda ronda de revisión.

Congresos internacionales y workshops (4):

1. P. Gómez-Abajo, E. Guerra, y J. de Lara. Wodel: a domain-specific language for model mutation. En *Proceedings of the 31st ACM/SIGAPP Symposium on Applied Computing, SAC*, páginas 1968–1973. ACM, 2016. Porcentaje de aceptación: 24,07 %.
2. P. Gómez-Abajo. A DSL for model mutation and its application to different domains. En *Proceedings of the Doctoral Symposium at the 19th ACM/IEEE International Conference of Model-Driven Engineering Languages and Systems, MoDELS*. ACM/IEEE, 2016.
3. P. Gómez-Abajo, E. Guerra, J. de Lara, y M. G. Merayo. Mutation Testing for DSLs (Tool Demo). En *Proceedings of the 17th ACM SIGPLAN International Workshop on Domain-Specific Modeling, DSM*, páginas 60–62. ACM, 2019.
4. P. Gómez-Abajo, E. Guerra, J. de Lara, y M. G. Merayo. Systematic engineering of mutation operators. En *32nd International Conference on Advanced Information Systems Engineering, CAISE Forum*, ACM, 2020.

Congresos nacionales (1):

1. P. Gómez-Abajo, E. Guerra, J. de Lara, y M. G. Merayo. Towards a model-driven engineering solution for language independent mutation testing. En *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, página 4pps. Biblioteca digital SISTEDES, 2018.

Presentaciones en cursos y seminarios internacionales (1):

1. P. Gómez-Abajo, E. Guerra, y J. de Lara. Wodel: a DSL for model mutation; and Wodel-Edu: its application to the automated generation of exercises. En *7th International Summer School on Domain-Specific Modeling, DSM-TP*, 22-26 Agosto, Ginebra, Suiza, 2016.

1.4. Financiación

Esta tesis se ha financiado con los proyectos Go-Lite (TIN2011-24139) y FLEXOR (TIN2014-52129-R) del Ministerio de Economía y Competitividad, y los proyectos SICO-MORo (S2013/ICE-3006) y FORTE (S2018/TCS-4314) de la Comunidad de Madrid.

1.5. Estructura de la memoria

A continuación se describe la estructura del resto de la memoria.

El **Capítulo 2** proporciona una introducción a la Ingeniería Dirigida por Modelos, y al trabajo relacionado de la mutación de modelos, la generación automática de ejercicios y las pruebas de mutación.

El **Capítulo 3** describe el DSL `WODEL` y las funcionalidades proporcionadas por la herramienta que se ha desarrollado para trabajar con este lenguaje. También se analiza la concisión de `WODEL` para expresar programas de mutación.

En el **Capítulo 4** se explica la herramienta `WODEL-EDU` para generación automática de ejercicios. También se incluye una evaluación con usuarios para medir la calidad de los ejercicios generados con la herramienta en el dominio de los autómatas finitos.

El **Capítulo 5** describe la herramienta `WODEL-TEST` para la creación de entornos de pruebas de mutación y su aplicación en el desarrollo de entornos de pruebas de mutación para un lenguaje de programación (Java) y un lenguaje de transformación de modelos (ATL), ambos lenguajes de uso extendido. El capítulo también incluye la evaluación de las dos herramientas que hemos generado.

En el **Capítulo 6** se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Conceptos Previos y Trabajo Relacionado

En este capítulo se describen los conceptos fundamentales sobre MDE necesarios para realizar este trabajo (Sección 2.1). Posteriormente, se realiza una revisión de la aplicación de las técnicas de mutación (Sección 2.2.1), así como de herramientas existentes para pruebas de mutación (Sección 2.2.2) y se describen algunos lenguajes utilizados para la definición de operadores de mutación (Sección 2.2.3). A continuación, se presenta el trabajo relacionado con la generación automática de ejercicios (Sección 2.3). Finalmente, en la Sección 2.4 se indican unas breves conclusiones del estudio que motivan la realización de esta tesis.

2.1. Conceptos previos sobre MDE

En esta sección se hace una breve revisión de las técnicas, y conceptos de MDE utilizados en este trabajo, tanto en el diseño e implementación del DSL `WODEL`, como en el de sus extensiones `WODEL-EDU` y `WODEL-TEST`.

2.1.1. Ingeniería dirigida por modelos

La Ingeniería Dirigida por Modelos es un paradigma de la Ingeniería del Software que propone la utilización activa de modelos y sus transformaciones a través de todas las etapas del ciclo de desarrollo del software [24]. En MDE, los modelos se utilizan para especificar, probar, simular y generar código para la aplicación final. A través del uso de modelos, este paradigma permite elevar el nivel de abstracción y de automatización en la construcción de software, y con ello atacar el principal problema en la creación de software: la gestión de la

complejidad; además de permitir mejorar diferentes aspectos de la calidad del software como la productividad y el mantenimiento [90].

En el ámbito de la Ingeniería del Software, la utilización de MDE permite organizar la información de una forma estructurada, lo que facilita la comprensión del proceso que se representa. MDE tiene como efecto inmediato que el ingeniero de software puede centrarse en la lógica del sistema que está creando, liberándose de aquellas tareas más mecánicas y repetitivas del desarrollo software, que mediante la utilización de técnicas de MDE quedan en gran medida automatizadas.

En el contexto de MDE, un sistema se define como “concepto genérico para el diseño de una aplicación, plataforma, o artefacto software” [112]. Además, un sistema puede estar compuesto de otros subsistemas, así como tener relaciones con otros sistemas (p. ej., un sistema se puede comunicar con otros).

En base a esta definición de sistema, un modelo es una abstracción de un sistema que existe en este momento o se pretende que exista en un futuro. Un modelo ayuda a definir un sistema y dar respuestas sobre el sistema estudiado sin la necesidad de utilizarlo directamente.

Los modelos se construyen con lenguajes de modelado, cuya definición en MDE viene dada por un meta-modelo. Un lenguaje de modelado es un lenguaje artificial que puede utilizarse para expresar una información, un conocimiento, o un sistema, en una estructura que queda definida por un conjunto de reglas coherentes. Estas reglas se utilizan para interpretar el papel de los elementos dentro de dicha estructura [24].

A su vez, un meta-modelo puede verse como un modelo que está expresado con un lenguaje que se denomina “lenguaje de meta-modelado” (p.ej., MOF). En este punto, uno se puede plantear con qué lenguaje se define el lenguaje de meta-modelado y así sucesivamente. A esta cuestión se puede responder con la noción de *arquitectura de cuatro niveles* (ver Figura 2.1) usada tradicionalmente para establecer la relación entre los distintos niveles de (meta-)modelado [90]:

- **M0. Nivel de datos del usuario o del mundo real:** caracteriza los datos del mundo real que son manipulados por el software. En este caso, el término “datos” está utilizado en un sentido amplio que incluye “procesos”, “conceptos”, “estados”, etcétera.
- **M1. Nivel de modelo:** caracteriza a los modelos que representan los datos del nivel M0.
- **M2. Nivel de meta-modelo:** caracteriza a los meta-modelos que describen los modelos que pueden construirse en el nivel M1.
- **M3. Nivel de meta-meta-modelo:** caracteriza a los meta-meta-modelos que describen los meta-modelos que pueden construirse en el nivel M2.

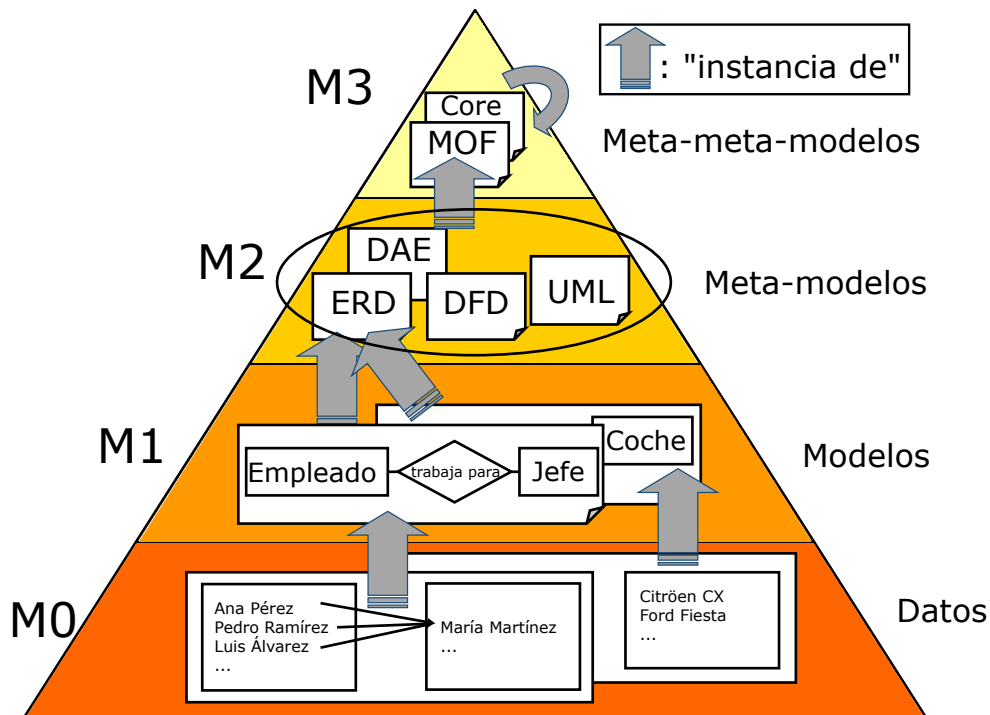


Figura 2.1 Pirámide de 4 niveles de MOF.

De acuerdo con esta arquitectura, un meta-meta-modelo se define con los mismos conceptos que el meta-meta-modelo define (definición circular).

Un meta-modelo describe la sintaxis abstracta de un lenguaje de modelado: los conceptos del lenguaje y las relaciones entre estos conceptos, así como las reglas que indican cuándo un modelo está bien formado. La técnica del meta-modelado consiste en la creación de modelos a partir de un lenguaje de modelado. El meta-modelo es la representación abstracta de las características que tiene este lenguaje. Para ello, típicamente se utilizan diagramas de clases. Por ejemplo, el Unified Modeling Language (UML) [94] establece que dentro de un modelo se pueden utilizar los elementos Clase, Atributo, Asociación, Paquete, etc.

A modo de ejemplo, la Figura 2.2 presenta un meta-modelo que describe autómatas finitos. Un autómata (objetos de tipo Automaton) consiste en un conjunto de estados (objetos de tipo State), transiciones (objetos de tipo Transition), y un alfabeto de símbolos (objetos de tipo Symbol). Un estado tiene un atributo name y puede ser inicial y/o final. Una transición conecta dos estados y puede tener vinculado un símbolo del alfabeto; las transiciones sin símbolo se consideran transiciones- λ . Una transición- λ indica que no es necesario procesar ningún símbolo para pasar del estado origen al estado destino.

Cuando es necesario incluir en un meta-modelo reglas que no se pueden expresar únicamente en términos de clases y relaciones, se utiliza el Object Constraint Language-

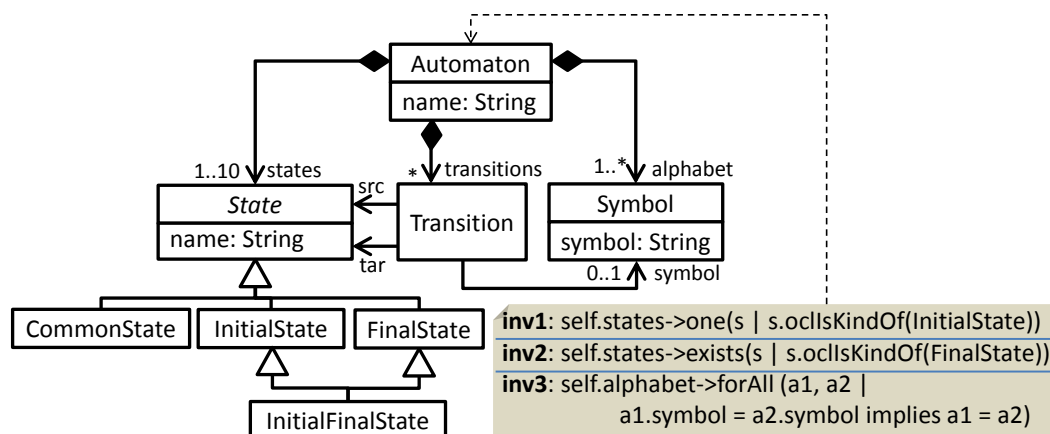


Figura 2.2 Meta-modelo para autómatas finitos.

ge (OCL) [94, 133], que es un lenguaje declarativo para describir restricciones de integridad para modelos (también llamadas invariantes). Aunque inicialmente OCL era únicamente una extensión formal a la especificación del lenguaje UML, en la actualidad forma parte de este estándar. OCL puede utilizarse con cualquier meta-modelo MOF [87], incluyendo UML.

Las restricciones OCL se definen en el contexto de una clase, y se evalúan en las instancias de dicha clase. El meta-modelo de la Figura 2.2 incluye tres restricciones OCL que cualquier objeto de tipo Automaton debe cumplir: la primera restricción requiere que haya exactamente un estado inicial, la segunda requiere que haya al menos un estado final, y la última requiere que los símbolos del alfabeto sean distintos.

Un modelo es una instancia de un meta-modelo, y se dice que es conforme a este meta-modelo si: los elementos del modelo son instancias de las clases del meta-modelo; las relaciones en el modelo son instancias de las asociaciones en el meta-modelo; y se satisface la cardinalidad en las asociaciones, las claves únicas, y las restricciones OCL incluidas en el meta-modelo.

La sintaxis abstracta consiste únicamente en la estructura de los datos, a diferencia de la sintaxis concreta, que incluye además información sobre la representación. La sintaxis concreta sirve para describir la notación del lenguaje de modelado. Esta sintaxis concreta puede ser gráfica o textual. Por ejemplo, una sintaxis concreta textual puede incluir características como los paréntesis (para agrupar), o comas (para listas), que no aparecen en la sintaxis abstracta, ya que están implícitos en la estructura.

La Figura 2.3 muestra un autómata finito representado como un modelo conforme al meta-modelo mostrado en la Figura 2.2 en la parte izquierda, y un ejemplo de sintaxis concreta gráfica del autómata finito que define en la parte derecha. Este autómata tiene dos estados: el estado q0, que es inicial y final; y el estado q1, que no es inicial ni final. Además,

este autómata tiene 4 transiciones: Si el autómata está en el estado q_0 y recibe el símbolo 0, permanece en el estado q_0 ; y si recibe el símbolo 1, pasa al estado q_1 . Si está en el estado q_1 y recibe el símbolo 0, pasa al estado q_0 ; y si recibe el símbolo 1, se mantiene en el estado q_1 .

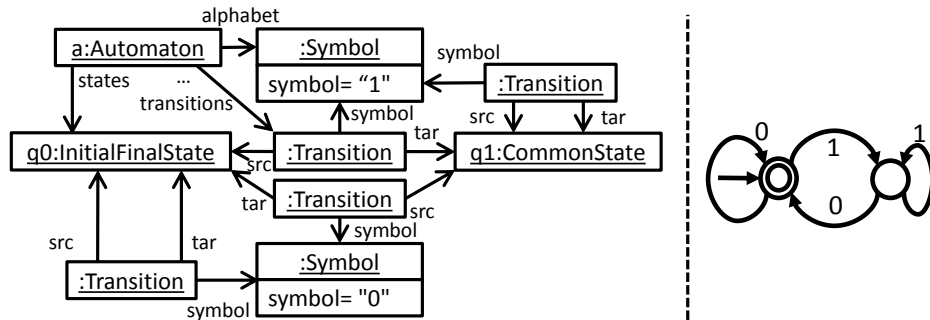


Figura 2.3 Modelo de autómata finito que acepta los números binarios pares.¹

Además este modelo de autómata cumple las tres restricciones OCL del meta-modelo representado en la Figura 2.2: hay exactamente un estado inicial (q_0), hay al menos un estado final (q_0), y los símbolos del alfabeto son distintos (0 y 1).

2.1.2. Lenguajes de dominio específico

Un Lenguaje de Dominio Específico (DSL) es un lenguaje de programación o especificación dedicado a resolver un problema en concreto, representar un problema específico, y proporcionar una técnica para solucionar una situación particular [86]. El concepto no es nuevo, pero se ha vuelto más popular debido al aumento de la utilización de modelado de dominio específico. Con la implementación de un DSL no se pretende proporcionar características para resolver cualquier tipo de problema, y seguramente con un DSL no se pueden implementar todos los programas que se pueden hacer con, p. ej., Java o C (que se denominan Lenguajes de Propósito General). Sin embargo, la hipótesis es que un DSL permite ser más eficiente a la hora de representar o resolver un problema en el dominio para el que ha sido diseñado.

Algunos ejemplos de DSL de uso extendido son: R^2 para estadística; Mata³ para programación matricial; Mathematica⁴ para matemáticas, fórmulas de hojas de cálculo y macros; y SQL [111] para consultas a bases de datos relacionales.

¹Para favorecer la legibilidad, se han omitido algunas referencias entre el objeto Automaton y el resto de objetos.

²<https://www.r-project.org/>

³<http://www.stata.com/>

⁴<http://www.wolfram.com/mathematica/>

Del mismo modo, podemos encontrar DSLs de modelado orientados a una tarea específica o dominio, que incluyen sus primitivas principales y abstracciones [67, 132]. Algunos ejemplos de este tipo de lenguajes incluyen BPMN [8] para el modelado de procesos de negocio, WebML [28] para el modelado de aplicaciones web, los Diagramas de Flujo de Datos (DFD) [121] para la especificación del flujo de datos de entrada y salida en un sistema o proceso, o ThingML [58] para la definición y generación de sistemas reactivos distribuidos como la Internet de las Cosas o los sistemas ciber-físicos. Los DSLs de modelado son útiles en áreas y disciplinas muy diversas, como la biología, la física, la gestión, o la educación, en las que los expertos de dominio pueden no ser ingenieros informáticos, o tener conocimientos en las plataformas y herramientas de MDE.

Los DSLs están adquiriendo cada vez mayor importancia en la Ingeniería del Software. Las herramientas para crear entornos para DSLs son también cada vez mejores, de forma que se puede desarrollar un DSL con relativamente poco esfuerzo. Estas herramientas permiten diseñar DSLs con sintaxis concreta gráfica o textual. Por ejemplo, algunas herramientas para crear DSLs gráficos son GMF [54], Graphiti [2] y Sirius [40]; y algunas herramientas para crear DSLs textuales son Xtext [18], EMFText [42] y Spoofax [116]. En la Sección 2.1.5.2 se describe más en detalle la herramienta Xtext, que es la que hemos utilizado para implementar el DSL `WODEL`.

Crear un DSL (con software que lo soporte) adquiere todo el sentido cuando permite que puedan expresarse tipos particulares de problemas o soluciones más claramente que con otros lenguajes existentes, y el tipo de problema se representa completamente. En los DSLs, la semántica del lenguaje queda muy cerca del dominio del problema para el que está diseñado. Tienen un alto nivel de abstracción al usuario, por tanto, están dirigidos a expertos en el dominio.

Existen dos enfoques principales para dotar de semántica a un DSL: operacional y denotacional. En el enfoque operacional, el significado de un programa o modelo bien formado es la traza de pasos de cómputo que resultan de su ejecución o simulación. En el enfoque denotacional, el significado de un programa o modelo se da en términos de otro formalismo cuya semántica está bien definida [107]. En este trabajo, se ha elegido un enfoque denotacional para dotar a `WODEL` de semántica mediante su compilación a código Java.

2.1.3. Transformación de modelos

La transformación de modelos representa otro ingrediente esencial de MDE que permite la definición de mapeados entre modelos diferentes. Las transformaciones se definen a nivel de meta-modelos, y se aplican sobre modelos que son conformes a esos meta-modelos.

Una transformación recibe uno o más modelos de entrada, y genera uno o más modelos de salida [24].

La transformación de modelos puede utilizarse con muy diversos propósitos. Por ejemplo, se puede usar para realizar análisis de modelos mediante su transformación a un lenguaje formal, optimización del modelo, rediseño, refactorización, simulación, etc. Esta transformación de modelos puede ser de distintos tipos: *in-place*, en la que los cambios se aplican dentro del propio modelo de entrada; o *out-place*, en la que se convierte un modelo de entrada conforme a un meta-modelo, a un modelo de salida conforme a otro (o a veces al mismo) meta-modelo.

En la Figura 2.4 se muestra el proceso que sigue una transformación *out-place* definida desde un meta-modelo A (meta-modelo de entrada de la transformación) a un meta-modelo B (meta-modelo de salida de la transformación). El proceso se aplica sobre el modelo A que es conforme al meta-modelo A. Al aplicar las reglas incluidas en el programa de transformación de modelos se obtiene un modelo B conforme al meta-modelo B de salida. En este proceso, el programa de transformación de modelos puede guardar un modelo de las trazas, que es la tabla de correspondencias entre los objetos del modelo A de entrada y los objetos en los que se han transformado dentro del modelo B de salida.

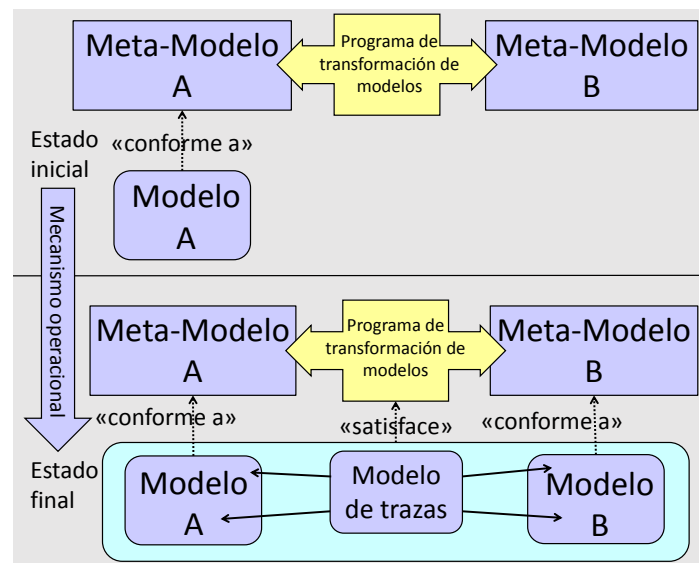


Figura 2.4 Transformación de modelos *out-place*.

Aunque una transformación de modelos podría implementarse con un lenguaje de programación de propósito general, como Java, diversos investigadores han propuesto lenguajes de dominio específico para definir transformaciones de modelos. Estos lenguajes de transformación de modelos pueden ser de distintos tipos [90]:

- **Declarativos basados en reglas:** Definir una transformación usando un lenguaje declarativo consiste en especificar las relaciones que los elementos de los modelos de entrada y salida deben satisfacer. Así, el motor de transformaciones construye el modelo de salida añadiendo los elementos necesarios al modelo de entrada para que la relación se cumpla. La adopción de un estilo declarativo facilita el mantenimiento de trazas entre los modelos de entrada y salida, pues dichas trazas están implícitas en las reglas que componen la transformación. El estándar QVT Relations [102] es el ejemplo por excelencia de los lenguajes que adoptan una aproximación declarativa para el desarrollo de transformaciones.
- **Imperativos:** En una transformación imperativa es la estructura del modelo de entrada la que dirige la transformación. Mientras se recorre el modelo de entrada se van creando los elementos del modelo de salida, a los que se les añaden los atributos y referencias correspondientes. En este sentido, los lenguajes de transformación imperativos, como QVT Operational Mappings [102], se parecen más a la programación con lenguajes de propósito general como Java.
- **Híbridos:** Los lenguajes de transformación híbridos combinan el estilo declarativo e imperativo. Aunque el estilo declarativo puede resultar más intuitivo (se especifica el qué, no el cómo), la utilización en ocasiones del estilo imperativo simplifica la tarea de desarrollar transformaciones especialmente cuando hay que definir comportamiento complejo. Un ejemplo de lenguaje de transformación híbrido es ATL (Atlas Transformation Language) [65].

La mutación de modelos es una técnica que está muy relacionada con la transformación de modelos. Del mismo modo que en la transformación *out-place* de modelos, en la mutación de modelos se parte de uno o varios modelos de entrada, que en la mutación de modelos son los modelos *semilla*, y se generan uno o varios modelos de salida, que en la mutación de modelos son los modelos *mutantes*. Por tanto, la mutación de modelos es un caso especial de transformación de modelos, en la que los modelos de entrada y los modelos de salida son conformes al mismo meta-modelo, y el programa de mutación especifica los cambios o *mutaciones* que se van a introducir en los modelos. En esta tesis se ha implementado un lenguaje de mutación (ver Sección 3) y un registro de mutaciones (ver Sección 3.2.2), que es la traza de los cambios entre el modelo semilla y los modelos mutantes generados. En la Figura 2.5 se muestra un esquema de la mutación de modelos.

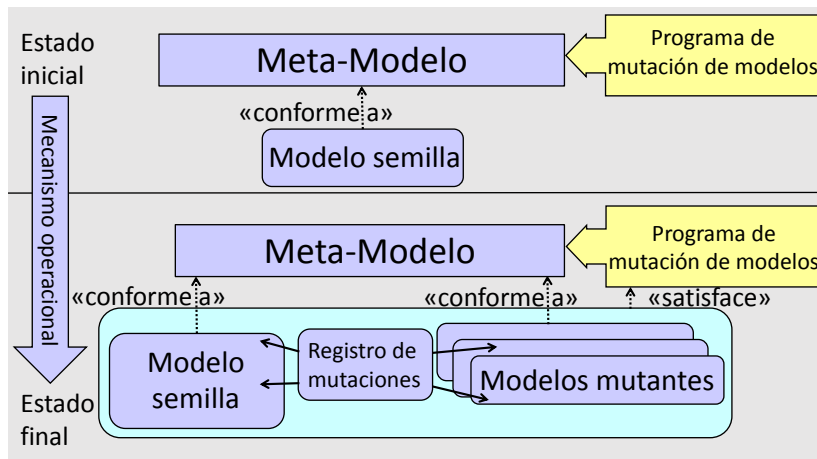


Figura 2.5 Mutación de modelos.

2.1.4. Generación de texto

La generación de texto es un tipo especial de transformación de modelos mediante la cual se genera un artefacto textual a partir de un modelo. Un caso particular de generación de texto es la generación de código, que tiene como objetivo producir código de un nivel más bajo de abstracción a partir de un modelo, típicamente para crear todo o parte de una aplicación, de forma muy similar a la que utilizan los compiladores para producir ficheros binarios ejecutables partiendo de código fuente. En este sentido, los generadores de código son a veces denominados *compiladores de modelos* [24].

Esta generación se hace normalmente por medio de un motor de plantillas basadas en reglas (ver Figura 2.6). De este modo, un generador de código consiste en un conjunto de plantillas con controles de contenido que cuando se aplican (se instancian) sobre los elementos del modelo, producen código. Algunos ejemplos de lenguajes de plantillas para generación de código son JET [63], Acceleo [3] y Xtend [18]. En la Sección 2.1.5.2 se describe el lenguaje de plantillas Xtend, que es el que hemos utilizado en este trabajo para generar el código Java a partir de las instrucciones de mutación programadas en el lenguaje WODEL.

Una vez se ha generado el código, si es necesario pueden utilizarse las herramientas habituales del IDE con el que se esté trabajando para optimizar el código fuente producido durante la generación, compilarlo, y finalmente desplegarlo.

2.1.5. Tecnologías utilizadas

Esta sección presenta el Eclipse Modeling Framework (EMF), que es la tecnología de modelado que se ha utilizado para representar modelos y meta-modelos; el framework Xtext

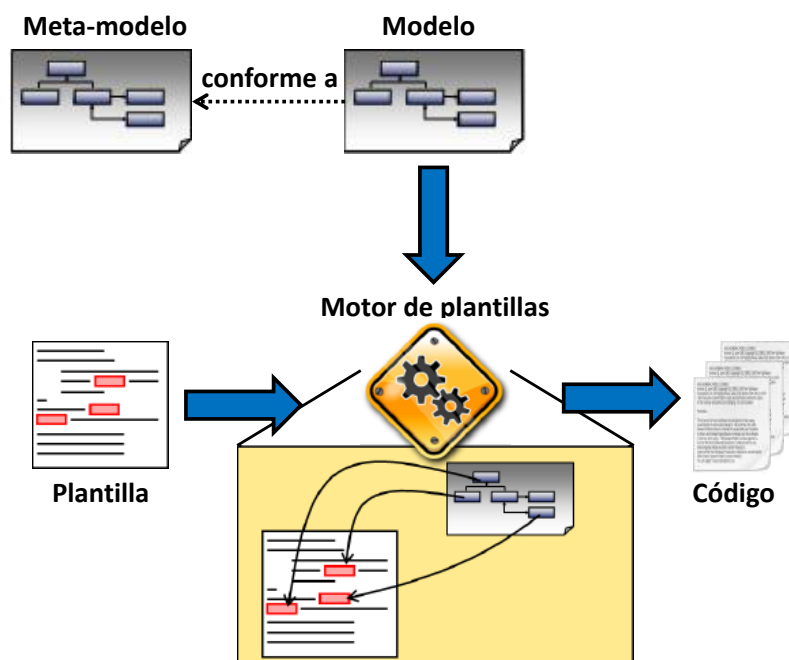


Figura 2.6 Arquitectura de un generador de código.

empleado para definir la sintaxis concreta textual del DSL `WODEL`; y el lenguaje de plantillas Xtend, tecnología utilizada para la generación de código Java a partir de los programas `WODEL`.

2.1.5.1. Eclipse Modeling Framework

El Eclipse Modeling Framework (EMF) [118] es un framework de modelado en Eclipse que facilita la construcción de herramientas y aplicaciones basadas en modelos. La forma canónica que proporciona para definir modelos de dominio (esto es, meta-modelos en terminología MDE) es utilizando XMI (XML Metadata Interchange), aunque también admite otros formatos como interfaces Java anotadas y diagramas UML. A partir de cualquiera de estas representaciones se pueden generar las otras dos, y también la correspondiente implementación de las clases en Java para poder editar modelos programáticamente. EMF es el estándar “de facto” de modelado y una implementación de MetaObject Facility (MOF) de la OMG.

EMF facilita una programación eficiente. Responde a la pregunta que el ingeniero del software se hace habitualmente: “¿Debo modelar o debo programar?”, con esta respuesta: “Puedes elegir cualquiera de las dos”.

Con EMF el modelado y la programación se pueden considerar equivalentes. En lugar de forzar una separación entre la ingeniería de alto nivel del modelado, y la programación de

bajo nivel, consigue que sean dos partes bien integradas del mismo trabajo. Muchas veces, especialmente con aplicaciones grandes, este tipo de separación es deseable, pero con EMF es el ingeniero del software quien decide el grado en el que se da esta separación.

El formato utilizado para representar meta-modelos en EMF es Ecore, que es a su vez un modelo EMF. Ecore es un subconjunto simplificado de UML. Ecore permite definir los elementos de un meta-modelo mediante clases Ecore (p. ej. *EClass* para representar la clase modelada, con un nombre, cero o más atributos, y cero o más referencias; *EAttribute*, para representar un atributo modelado, que tiene un nombre y un tipo primitivo; *EReference*, para representar un extremo de una asociación entre clases, con un nombre, un valor booleano que indica si es una referencia contenedora, y la clase destino de la referencia; etc.). Ecore incluye además soporte para la definición y evaluación de restricciones OCL.

Un proyecto EMF de Eclipse genera de forma automática las clases Java necesarias para manejar los modelos conformes al meta-modelo Ecore, y se puede añadir la funcionalidad adicional que se desee. EMF proporciona una API [118] mediante la que se manejan los objetos EMF de forma genérica. Utiliza la representación canónica XMI (XML Metadata Interchange) para la serialización directa de los modelos Ecore y sus instancias, de forma que no se almacena ninguna información innecesaria.

2.1.5.2. Xtext y Xtend

Xtext [18] es un framework de Eclipse de código abierto para implementar DSLs textuales e integrarlos con el IDE de Eclipse. Xtext permite implementar lenguajes rápidamente, cubriendo todos los aspectos de un entorno completo de edición del lenguaje: analizador sintáctico, generador de código o intérprete, coloreado de sintaxis, completado de código, marcadores de errores, infraestructura automática de compilado, etc. Para especificar un DSL en Xtext basta con crear un proyecto Xtext a partir del meta-modelo del DSL, adaptar la gramática del DSL creada por defecto por Xtext, y completar, si fuese necesario, la funcionalidad del editor y también, la generación de código. Xtext está implementado a partir de EMF, lo que conlleva que cualquier programa de un DSL creado con Xtext es un modelo de EMF, y puede ser serializado como modelo XMI.

Xtend [18] es un lenguaje de programación parecido a Java capaz de interactuar completamente con el sistema de tipos de Java, y que se compila a código Java de forma automática. Tiene una sintaxis sencilla y compacta, y características avanzadas como inferencia de tipos o expresiones lambda. Xtext fomenta la utilización de Xtend, siendo Xtend el lenguaje empleado para configurar muchos de los servicios asociados al editor del DSL, como validaciones de los modelos, generadores de código, etc.

2.2. Técnicas de mutación

La revisión de trabajos relacionados con las técnicas de mutación se divide en tres partes. En primer lugar, se revisan trabajos recientes que han utilizado las técnicas de mutación con diferentes objetivos. En segundo lugar, se revisan trabajos que utilizan técnicas de mutación para realizar pruebas de mutación. Por último, se revisan lenguajes de dominio específico existentes para la definición de operadores de mutación.

2.2.1. Aplicaciones de las técnicas de mutación

En este apartado revisamos los trabajos relevantes sobre técnicas de mutación que hemos identificado, clasificados en trabajos de mutación de artefactos de cualquier tipo, no necesariamente un modelo, y en trabajos de mutación de modelos.

Las técnicas de mutación de carácter general se han utilizado en campos diversos como las pruebas de mutación de sistemas adaptativos [16], la prueba de programas [23, 64, 69, 81, 93, 113, 128], la prueba de fórmulas lógicas [60], la generación de operadores de mutación en algoritmos genéticos [134], y la reparación de restricciones OCL [29].

En el área de modelado, las técnicas de mutación se han aplicado con mayor frecuencia a la prueba de mutación de modelos de distinto tipo, tales como máquinas de estados [46, 61, 131], protocolos de comunicación [20], redes de Petri [44], líneas de productos software [59, 77], sistemas de transiciones con características [39], programas en C [131] o transformación de modelos [10, 68, 92, 126]. Otras aplicaciones donde también se ha aplicado la mutación de modelos son la generación de modelos [99], la generación de ejercicios [104], la generación de casos de prueba [6], la evaluación de métodos de detección de clones [119, 120], algoritmos genéticos [89], y la ingeniería del software basada en búsqueda [48, 122].

A continuación se revisan brevemente los trabajos de mutación mencionados en los dos párrafos anteriores, a excepción de los trabajos sobre pruebas de mutación, que se revisarán en la Sección 2.2.2, y los de generación de ejercicios, que se revisarán en la Sección 2.3.

En cuanto a los algoritmos genéticos, Moawad et al. [89] describen la aplicación de técnicas de diseño de MDE para definir las funciones de fitness y los operadores de mutación en los algoritmos genéticos multiobjetivo (MOEAs), de forma que no es necesario tener conocimiento de la codificación MOEA. Proponen aprovechar las técnicas de MDE para que los desarrolladores sólo tengan que preocuparse de la lógica para resolver el problema, independientemente del dominio del mismo. Proponen tres operadores de mutación: AddInstance, RemoveInstance, y ChangeWeight. Como caso de estudio, muestran una aplicación de su trabajo en un problema de distribución de tareas de una serie de aplicaciones en la nube.

Una vez evaluado el caso de estudio demuestran que la solución aportada que utiliza técnicas MDE tiene un rendimiento similar a las soluciones existentes ad-hoc.

Woodward y Swan [134] proponen una generación automática de operadores de mutación para algoritmos genéticos. Implementan los operadores de mutación utilizando máquinas de registros: suma, incremento, decremento, inversión del valor, puesta a 0, asignación de un valor aleatorio y asignación de valor fijo. También seleccionan los operadores de mutación con los que se obtienen mejores resultados al aplicar el algoritmo genético.

En cuanto a la aplicación de las técnicas de mutación a la evaluación de la calidad en la detección de clones de modelos, Stephan et al. [119, 120] proponen la herramienta MuMonDE. La mecánica utilizada es parecida a la de las pruebas de mutación. Se generan mutantes de los modelos originales y posteriormente se utiliza la herramienta de detección de clones comparando estos mutantes con el correspondiente modelo original. A partir de aquí se puede comprobar si la herramienta detecta correctamente las variaciones, y así se puede medir la calidad de la herramienta de detección de clones. En este trabajo incluyen los operadores de mutación de cambio de la presentación y orden de los elementos, modificación de su valor, y otros cambios de la estructura de los modelos. En estos trabajos se evalúa la herramienta SIMONE de detección de clones de modelos de Simulink.

Clariso y Cabot [29] utilizan las técnicas de mutación para realizar una reparación automática de errores en restricciones OCL. Las reparaciones se implementan utilizando técnicas de mutación que hacen que la restricción OCL sea más o menos restrictiva que la original. La aplicación de las reparaciones puede ser automática o permitiendo al usuario elegir entre varias opciones.

Las técnicas de mutación se utilizan también en la ingeniería del software basada en búsqueda [48, 122]. Fleck et al. [48] proponen la transformación de modelos basada en búsqueda. Para implementar estas transformaciones utilizan las herramientas MoMoT y Henshin. MoMoT proporciona facilidades para manejar las instancias de estos problemas a través de transformaciones de modelos modeladas como reglas de transformación de grafos en Henshin. Las características deseadas y prohibidas (objetivos y restricciones) del modelo de salida se pueden especificar utilizando OCL o un lenguaje de expresiones tipo Java (Xbase). Las técnicas de optimización basadas en búsqueda pueden utilizarse para buscar el conjunto de transformaciones óptimo de Pareto, es decir, una secuencia ordenada de transformaciones y sus parámetros, para producir modelos con estas características.

A modo de resumen, las características principales de estos trabajos se presentan en las Tablas 2.1, 2.2 y 2.3. Estas tablas muestran el nombre de la herramienta (si la hay) y los autores del artículo que presenta el enfoque, el dominio del problema que abordan, la aplicación de las técnicas de mutación, el artefacto que se muta, un resumen de los operadores de mutación

que definen, y el lenguaje utilizado para crearlos. Se puede observar que un total de 24 de los 31 trabajos revisados corresponden a pruebas de mutación [6, 7, 10, 16, 20, 23, 39, 44, 46, 59–61, 64, 68, 69, 77, 81, 92, 93, 99, 113, 131, 126, 128]. Aunque en la literatura se encuentran también trabajos en los que se mutan otros tipos de artefactos, p. ej., código, árbol sintáctico, etc., en esta revisión nos hemos centrado en aquellos en los que se mutan modelos, de los que hemos seleccionado estos 19 [7, 10, 20, 39, 44, 46, 59, 68, 77, 89, 92, 93, 99, 104, 119, 120, 122, 126, 128]. Los operadores de mutación de estos trabajos son fijos en la mayoría de los casos, permitiendo su configuración únicamente en tres de ellos: transformaciones de modelos basadas en búsqueda en MoMoT [48], pruebas de mutación para C en MILU [64], y pruebas de mutación para Java en Major [66].

Como se observa, todas estas propuestas son específicas para un dominio y aplicación. `WODEL` puede resultar útil para superar estas limitaciones de forma que permite la reutilización y extensión de las mutaciones.

2.2.2. Pruebas de mutación

A continuación se presenta una breve introducción a las pruebas de mutación (*Mutation Testing*, MT). Como se describirá en el Capítulo 5, la extensión de `WODEL` para la creación automática de entornos de pruebas de mutación `WODEL-TEST` se fundamenta en estos conceptos teóricos, junto con otros propios de MDE. Se incluye este apartado en esta Sección 2.2.2 ya que se ha considerado útil para la comprensión del trabajo relacionado de pruebas de mutación.

El objetivo de las pruebas de mutación es evaluar la calidad de un conjunto de pruebas, es decir, medir su efectividad con respecto a su habilidad para detectar fallos [37, 57]. La Figura 2.7 describe el esquema de dicho proceso.

En primer lugar, dado un programa se introducen pequeños cambios sintácticos para generar programas llamados *mutantes*. Los mutantes se generan por medio de *operadores de mutación* que simulan errores comunes cometidos por programadores competentes. Se distingue entre los mutantes de *primer orden*, si se obtienen aplicando un operador de mutación una vez al programa original, y mutantes de *orden superior*, si obtienen tras la aplicación de varios operadores de mutación.

A continuación, el conjunto de pruebas se aplica al programa original y a los mutantes. Si la salida producida por un mutante es diferente de la salida producida por el programa original para algún test en el conjunto de pruebas, entonces se dice que se ha *matado* el mutante; en otro caso, es un mutante *vivo*. Algunos mutantes vivos pueden a su vez ser *equivalentes* al programa original, en el sentido de que ningún caso de prueba puede matarlos. Por ejemplo, un operador que añade un “+ 0” a una expresión aritmética tiene como resultado un mutante

Autores/Herramienta	Dominio	Aplicación	Artefacto mutado	Operadores de mutación	Lenguaje
Aicherning et al. [6]	Máquinas de estados	Generación de casos de prueba/ Pruebas de mutación	Modelo	Poner a falso protección de transición ^a , eliminar las acciones de entrada, y eliminar o modificar disparadores y eventos de señal	Semánticas OOAS [21]
Alhwikem et al. [7]	DSLs	Generación de operadores de mutación	Modelo EMF	Eliminación, modificación y creación de elementos	Lenguaje formal
Aranega et al. [10]	Transformación de modelos	Pruebas de mutación	Modelo del programa de transformación	Navegación, filtrado y modificación	Ecore
Bartel et al. [16]	Sistemas adaptativos	Pruebas de mutación	Política adaptativa	Incorporar propiedad, modificar acción de regla	Kermeta
Bombieri et al. [20]	Protocolos Transaction-Level Modeling (TLM)	Pruebas de mutación	Modelo de máquina de estados finitos correspondiente al TLM	Cambiar estado destino, habilitar o actualizar la función	Primitivas TLM
Bradbury et al. [23]	Java	Pruebas de mutación	Código fuente	Operaciones de mutación de concurrencia	TXL [32]
Clarisó y Cabot [29]/ EMFtoCSP	OCL	Reparación de OCL	Árbol sintáctico	Weakening y strengthening	Java, Prolog
Devroey et al. [39]/ VIBeS	Sistemas de transiciones con características (FTSs)	Pruebas de mutación	Modelo del FTS	StateMissing, ActionExchange y WrongInitialState	Pequeño DSL Java
Fabbri et al. [44]/ Proteum	Redes de Petri	Pruebas de mutación	Modelo de la red de Petri	Eliminación, creación, traslado, e intercambio de entrada y salida cambio de la marca inicial	C
Fabbri et al. [46]/ Proteum	Máquinas de estados	Pruebas de mutación	Modelo de la máquina	Eliminación de transición, cambio de estado inicial, eliminación intercambio y creación de evento o de salida, creación de estado	C
Fleck et al. [48]/ MoMoT	Ingeniería del software basada en búsqueda	Transformaciones de modelos basadas en búsqueda	Modelos EMF	Configurables	MoMoT, Henshin
Henard et al. [59]	Ingeniería del software basada en búsqueda	Generación de casos de prueba/Pruebas de mutación	Fórmula del modelo de características	Negación de un literal, omisión de un literal, OR por AND	C++
Henard et al. [60]/ MutaLog	Fórmulas lógicas	Pruebas de mutación	Fórmula CNF	Omisión o negación de un literal, omisión o negación de una cláusula, AND por OR y viceversa	Java

Tabla 2.1 Resumen de los trabajos de mutación analizados (a).

^aEsto puede llevar a que la transición se transforme en un bucle automático, dado que el modelo “traga” el evento que la desencadena.

Autores/Herramienta	Dominio	Aplicación	Artefacto mutado	Operadores de mutación	Lenguaje
Hierons y Merayo [61]	Máquinas de estados	Pruebas de mutación	Máquinas de estado probabilísticas y estocásticas	Cambio estado inicial, modificación de probabilidad de transición, cambio de estado destino y creación de una transición	Lenguaje formal
Jia y Harman [64]/ MLU	C	Pruebas de mutación	Árbol sintáctico	Configurables	Mutation Operator Constraint Script
Khan y Hassine [68]	Transformación de modelos	Pruebas de mutación	Modelo	MatchedRule a LazyRule y viceversa, eliminación y adición de binding, eliminación y adición de condición en filtro, cambio de clase en filtro y en binding, cambio del modo de ejecución, eliminación de instrucción return	Java
Kim et al. [69]	Java	Pruebas de mutación	Código intermedio	Reemplazo de tipos, cambio de orden, modificación y eliminación de argumentos, eliminación de métodos.	Java
Lackner y Schmidt [77]	Líneas de producto software	Pruebas de mutación	Modelos de mapeado/ Modelos de características/ Máquinas de estado UML	Operadores de mutación básicos de modelos de mapeado/ de modelos de características/ de máquinas de estado UML	Lenguaje formal
Ma et al. [81]	Java	Pruebas de mutación	Clases Java mediante reflexión	Modificación de la visibilidad, herencia, polimorfismo, sobrecarga.	Java
Polymer Moawad et al. [89]	Algoritmos genéticos multi-objetivo (MOEAs)	Algoritmos genéticos	Modelo del problema MOO (optimización multi-objetivo)	AddInstance, RemoveInstance y ChangeWeight	Java
Mottu et al. [92]	Transformación de modelos	Pruebas de mutación	Modelo de la transformación	Cambio de relacion, cambio en parámetros de filtrado, reemplazo de clase por una creada compatible, y creación y eliminación de asociación de clase	Kerneta, Tefkat, Java
Nguyen et al. [93]	Delegación de permisos	Pruebas de mutación	Modelo de sistema de gestión de bibliotecas (LMS)	Delegación de permiso, delegación de rol, delegación monotonica, y delegación basada en contexto, o en rol o permiso específico	Java
René [66]/ Major	Java	Pruebas de mutación	Árbol sintáctico	Configurables	Major-Mml

Tabla 2.2 Resumen de los trabajos de mutación analizados (b).

Autores/Herramienta	Dominio	Aplicación	Artefacto mutado	Operadores de mutación	Lenguaje
Pietsch et al. [99]/ SiDiff	Generación de modelos	Prueba de programas que manipulan modelos	Modelo EMF	Creación de Package, Class, Interface, Attribute, Method, Parameter, Association	Ecore
Sadigh et al. [104]	Máquinas de estados	Generación automática de ejercicios	Modelo	Cambiar estado inicial, cambiar estado destino de una transición, crear nueva transición, añadir/eliminar estados	Promela
Simão y Maldonado [113]/ MuDel	C	Pruebas de mutación	Árbol sintáctico	Eliminación de instrucción, reemplazo de instrucción, “donothing”, “abort”	MuDel
Stephan et al. [119, 120]/ MuMonDE	Detección de clones de modelos Simulink	Calidad de la detección de clones por SIMONE	Modelo Simulink	Cambio de presentación y orden, y modificación de valor de elementos, cambios de estructura	Java
FitnessStudio Strüber [122]	Ingeniería del software basada en búsqueda	Generación de operadores de mutación eficaces	Modelo EMF	Se generan automáticamente a partir de reglas de transfor- mación de modelos de Henshin	Henshin, Java
Vinzenci et al. [131]/ Muta-Pro	Programas en C/Modelos de máquinas de estados	Pruebas de mutación	Código C	Sufficient Incremental Unit Testing Strategy-SUS [130]	MuDel
Troya et al. [126]	Transformación de modelos	Pruebas de mutación	Modelo	Creación y eliminación de regla, filtro, patrón de entrada/salida de regla y binding; cambio de nombre y patrón de entrada/salida de regla; cambio de condición de filtro y de patrón de entrada/salida de regla; y cambio de valor de característica de binding	ATL
Tuya et al. [128]/ SQLMutation	SQL	Pruebas de mutación	Modelo DOM de una representación XML	Modificación de principales cláusulas SQL, operadores en condiciones de la consulta y expresiones, valores NULL e identificadores	Java
Woodward y Swan [134]	Algoritmos genéticos	Generación de operadores de mutación para algoritmos genéticos	Algoritmo genético	Suma, incremento, decremento, inversión, puesta a 0, asignación de valor aleatorio o fijo	Java

Tabla 2.3 Resumen de los trabajos de mutación analizados (c).

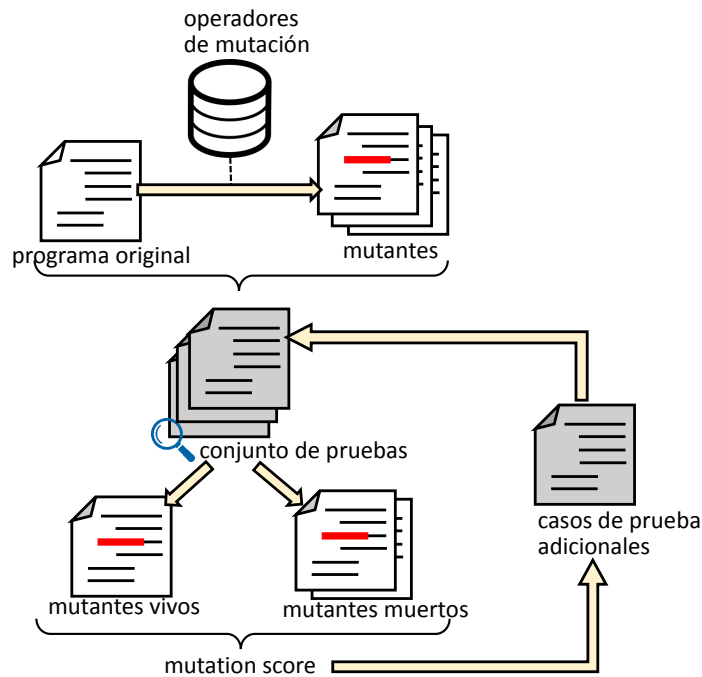


Figura 2.7 Proceso de pruebas de mutación.

equivalente, dado que el mutante es sintácticamente diferente al programa original, pero es semánticamente equivalente.

La utilidad de las pruebas de mutación se basa en la creencia de que, si un conjunto de pruebas es bueno distinguiendo un programa de sus mutantes, entonces probablemente es bueno descubriendo fallos reales. La adecuación del conjunto de pruebas se determina por el *mutation score* que corresponde a la proporción de mutantes que se han matado sobre el número total de mutantes no equivalentes. Si el *mutation score* se considera bajo, entonces debe mejorarse el conjunto de pruebas con nuevos casos de prueba capaces de matar los mutantes vivos y por tanto mejorar el *mutation score*.

Las pruebas de mutación se aplicaron inicialmente a código fuente, y por tanto, existen catálogos de operadores de mutación para lenguajes de programación como Ada [96], C [5, 64], C++ [35, 76], Fortran [70], Java [23, 69, 81] o SQL [128]. Posteriormente, las pruebas de mutación se han aplicado a otros artefactos como especificaciones formales (p. ej., máquinas de estado finitas [46, 61], diagramas de estados [45, 125], redes de Petri [44], CSP [117]) o servicios web [43, 79]. Por tanto, existe un amplio abanico de herramientas de pruebas de mutación para dar soporte al análisis de mutación para lenguajes diferentes. Inicialmente, estas herramientas se desarrollaron para trabajar con las técnicas de pruebas de mutación en el entorno académico [34, 36], pero desde el comienzo de los años 2000,

muchas herramientas tienen como objetivo conseguir que las pruebas de mutación puedan aplicarse de manera práctica a aplicaciones reales [64, 73, 82, 108, 127].

La relevancia de las pruebas de mutación se demuestra por la gran cantidad de sistemas software y dominios en los que se han usado, incluyendo aplicaciones web [88], software crítico de seguridad [15], diagramas de bloques de función [110], entornos en la nube y HPC [27], aplicaciones Android [38], interfaces gráficas de usuario [9], protocolos IoT [14] o fallos de memoria [135].

En las Tablas 2.1, 2.2 y 2.3 las pruebas de mutación son la principal aplicación de las técnicas de mutación. Existen herramientas de pruebas de mutación para los lenguajes de programación más utilizados, como Java [23, 69, 81], C [5, 64] o SQL [128]. También se han aplicado a notaciones de modelado con semánticas ejecutables, como redes de Petri [44] y máquinas de estados [46, 61].

Varios autores han aplicado las pruebas de mutación en el área de MDE, especialmente a transformaciones de modelos. Por ejemplo, en [92] se define una colección de operadores de mutación genéricos basados en las actividades principales involucradas en una transformación de modelos (navegación, filtrado, creación del modelo de salida y modificación del modelo de entrada). Otros trabajos proponen enfoques sistemáticos para generar operadores de mutación ya sea para un lenguaje específico [68, 126] o para varios lenguajes [7]. Por último, también se han propuesto operadores de mutación de transformaciones que emulan los errores que cometen los programadores según estudios empíricos [56].

El análisis de mutaciones se ha aplicado a la generación de casos de prueba adecuados (es decir, modelos) para probar transformaciones de modelos [10, 109]. Curiosamente, en el trabajo de Sen y Baudry [109], los operadores son reglas de transformación de grafos derivadas automáticamente del meta-modelo que se está transformando.

La técnica de las pruebas de mutación también se ha utilizado con líneas de producto software [59, 77]. Henard et al. [59] proponen una generación automática de casos de prueba para líneas de producto software. El artefacto mutado en este caso son los modelos de características de las líneas de producto software, y los operadores de mutación que proponen consisten en la realización de cambios en la fórmula lógica de estos modelos de características. Lackner y Schmidt [77] desarrollan un sistema de mutación para estimación de la calidad de los conjuntos de prueba de líneas de productos software. Definen operadores de mutación para modelos de características, máquinas de estado UML, y modelos de mapeado. Describen tres casos de estudio: una tienda online, una máquina expendedora de billetes, y un sistema de alarma.

La mayoría de las veces, las herramientas de pruebas de mutación se crean a mano sin tener en mente la posibilidad de extenderlas, y además no se habilita la definición personalizada

de conjuntos de operadores de mutación. Por otro lado, construir una herramienta de pruebas de mutación supone una gran inversión en términos de esfuerzo. Esto puede obstaculizar la utilización de las pruebas de mutación en lenguajes emergentes o en DSLs [132] con un número menor de usuarios, ya que sin automatización, el desarrollo de herramientas de pruebas de mutación para estos lenguajes puede no resultar rentable. Finalmente, puede existir mayor complejidad en el desarrollo si la herramienta de pruebas de mutación requiere manejar tecnologías heterogéneas (p. ej., una herramienta de pruebas de mutación para JavaScript puede requerir utilizar páginas HTML y documentos CSS a su vez).

La herramienta *WODEL-TEST* incluida en esta tesis supone una contribución a la comunidad para superar estos inconvenientes.

2.2.3. Lenguajes de definición de operadores de mutación

La mayoría de los trabajos estudiados utilizan lenguajes de propósito general para especificar los operadores de mutación, como C [44, 46], C++ [59], Java [29, 60, 69, 81, 89, 92, 93, 122, 126, 128], Prolog [29], o el lenguaje de verificación de modelado Promela [104]. Aicherning et al. [6] utilizan semánticas OOAS, Bombieri et al. [20] utilizan primitivas TLM mutadas, y Bradbury et al. [23] utilizan el lenguaje de transformación de código TXL [32] con este mismo objetivo. Otros trabajos modelan los operadores de mutación utilizando un lenguaje de transformación de modelos [16, 92, 126], y dos de los trabajos revisados realizan las transformaciones mediante gramáticas de grafos [48, 122]. Los problemas que se identifican en todos estos trabajos es que son demasiados generales, no tienen primitivas específicas ni servicios útiles para mutación, como son el registro de las mutaciones aplicadas o la detección de mutantes equivalentes.

El trabajo de Alhwikem et al. [7] presenta un enfoque para generar operadores de mutación de forma automática. Estos operadores de mutación se aplican sobre meta-modelos Ecore. Sin embargo, los operadores de mutación generados sólo añaden, eliminan y modifican características de los elementos, y no sirven para crear o eliminar elementos, cuestión que sí es posible utilizando *WODEL*.

El trabajo de Strüber [122] presenta la herramienta FitnessStudio que genera de forma automática operadores de mutación para utilizarlos en la ingeniería del software basada en búsqueda. Los operadores de mutación se generan de la siguiente forma: (i) se consideran los modelos directamente, lo que permite modificar los candidatos a la solución utilizando reglas de transformación que se pueden modificar por medio de reglas de orden superior; (ii) el nivel inferior del framework incluye varios componentes configurados por el usuario: la función de fitness, el operador transversal, y las restricciones del problema, lo que permite generar completa y automáticamente el operador de mutación; y (iii) el nivel superior proporciona

el operador de mutación que utiliza una transformación de orden superior para modificar el operador de mutación de nivel inferior. La herramienta selecciona los operadores de mutación que tienen mayor eficacia. A modo de ejemplo, los autores muestran cómo generar operadores de mutación para las fórmulas lógicas de las líneas de productos software. Si bien este enfoque es interesante para automatizar todo el proceso de generación de operadores de mutación, resulta menos eficaz que utilizar un enfoque semi-automático como el que proporciona la herramienta *WODEL* (ver secciones 3.3.1 y 3.3.2).

Sólo algunas de las herramientas estudiadas utilizan un lenguaje de dominio específico creado para la definición de los operadores de mutación, como *MuDel* [113, 131], *MiLu* [64] y *Major* [66]. Los siguientes párrafos describen estos lenguajes.

El meta-lenguaje *MuDel* [113, 131] se utiliza para realizar mutaciones en programas en lenguaje C. En *MuDel* se pueden declarar variables y definir sustituciones. El Listado 2.1 muestra un ejemplo de la codificación en este meta-lenguaje del operador de mutación para eliminar una instrucción.

```
1 operator STDL
2   var :s [statement]
3 begin
4   * replace [statement<:s>]
5     by [statement<:>]
6 end operator
```

Listado 2.1 Programa *MuDel* para eliminar una instrucción.

Devroey et al. [39] utilizan un pequeño DSL Java para facilitar la aplicación de los operadores de mutación. El Listado 2.2 es un ejemplo de un programa en este lenguaje que carga un modelo TS a partir de un fichero XML y aplica el operador de mutación *ActionExchange* sobre dicho modelo.

```
1 TransitionSystem ts = loadTransitionSystem("model.ts");
2 MutationOperator op = actionExchange(ts)
3   .transitionSelectionStrategy(RANDOM)
4   .actionSelectionStrategy(RANDOM)
5   .done();
```

Listado 2.2 Programa en el DSL de Devroey et al. [39] para cambiar la acción de una transición.

MiLu proporciona un lenguaje de script llamado *Mutation Operator Constraint Script* (MOCS) [64] que permite la configuración de los operadores de mutación que se quieren aplicar sobre programas escritos en C. El Listado 2.3 presenta un ejemplo del operador de

mutación que sustituye un símbolo de igualdad (==) por uno de asignación (=). Este lenguaje sólo permite este tipo de instrucciones sencillas de reescritura.

```
1 "==" -> "="
```

Listado 2.3 Sustitución de == por = en el lenguaje MOCS de la herramienta MiLu.

La herramienta Major para realizar pruebas de mutación en programas Java [66] proporciona el DSL Major-Mml (que es en realidad un lenguaje de scripts) para crear operadores de mutación y configurar el proceso de mutación. El Listado 2.4 presenta un ejemplo del operador de mutación de reemplazo de un operador aritmético. Major-Mml se describe en detalle en la Sección 5.3.1 donde se evalúan las herramientas de pruebas de mutación de Java en comparación con un entorno de pruebas de mutación de Java creado con nuestra propuesta.

```
1 // Configuración del operador de mutación AOR
2 BIN (*) -> {/, %};
3 BIN (/) -> {*, %};
4 BIN (%) -> {*, /};
5
6 // Activación del operador
7 AOR ;
```

Listado 2.4 Programa en Major-Mml para reemplazar un operador aritmético.

Estos DSLs son específicos para un lenguaje (C, Java, etc.) y no dan soporte a funcionalidades útiles en un proceso de mutación como, por ejemplo, la detección de mutantes equivalentes y las trazas del proceso de mutación. La herramienta de Devroey et al. funciona a nivel del modelo, y MiLu y Major funcionan a nivel del AST. Como se podrá comprobar, el lenguaje `WODEL` es un enfoque unificado para definir y aplicar todos estos operadores de mutación, siempre que sea posible modelar el artefacto que se quiere mutar, y además proporciona de forma automática un conjunto de funcionalidades para el proceso de mutación que en otro caso habría que implementar de forma independiente en cada una de las herramientas.

2.3. Generación automática de ejercicios

Esta tesis presenta también la aplicación de `WODEL` a la generación automática de ejercicios mediante la creación de una herramienta que hemos llamado `WODEL-EDU`. Por este motivo se ha realizado un estudio de trabajos relacionados de generación y evaluación automática de ejercicios, encontrando principalmente trabajos en evaluación de ejercicios de

programación. Algunos sitios web de evaluación automática de ejercicios de programación son, por mencionar dos, *CodinGame*⁵ y *Acepta el reto*⁶.

Se ha mencionado en la Sección 1.1 el trabajo de Sadigh et al. [104]. En este trabajo, los autores utilizan técnicas de mutación para generar ejercicios de máquinas de estado para un curso MOOC. No obstante, [104] es un trabajo en desarrollo y el objetivo de los autores es construir el sistema a mano. Esto podría hacerse de forma automática con WODEL. Además, WODEL-EDU puede aplicarse para generar ejercicios en cualquier dominio.

Queirós y Leal [101] proponen la herramienta *Petcha (Programming Exercises TeaCHing Assistant)*. Petcha es una herramienta de Asistencia Automática a la Enseñanza en cursos de programación. El objetivo final de Petcha es incrementar el número de ejercicios de programación resueltos satisfactoriamente por los estudiantes. Petcha sirve de asistente para los profesores a la hora de crear los ejercicios de programación, desplegar los ejercicios en un repositorio y configurar la actividad de la programación en un LMS (*Learning Management System*). Desde el punto de vista de los estudiantes, Petcha permite seleccionar una actividad en el LMS y ejecutar la actividad utilizando el IDE.

En el artículo de Queirós y Leal se hace una revisión de varios sistemas de creación automática de ejercicios de programación, y también de otros sistemas de evaluación automática de ejercicios de este tipo. En este trabajo se indica que, a pesar de algunos intentos para definir un formato común de descripción de ejercicios de programación [49, 129], cada uno de estos sistemas tiene un formato propio.

Uno de estos sistemas es *JExercise* [124], un plugin para Eclipse al que hay que proporcionar la especificación de los elementos requeridos y el comportamiento deseado, un conjunto de pruebas *JUNIT* para verificar el código, y un modelo de la solución requerida.

En cuanto a los sistemas de evaluación automática de ejercicios, se indica que muchos de ellos proporcionan otras características, como soporte multi-lenguaje de programación, tipo de evaluación estática o dinámica, feedback, interoperabilidad, contexto de aprendizaje, seguridad y plagio. El enfoque habitual de este tipo de evaluación es de caja negra: se compila el código y se ejecuta el programa con un conjunto de casos de prueba que cuenta con ficheros de entrada y salida. El programa se cataloga como aceptado si compila sin errores, y además la salida de la ejecución de cada prueba es la misma que la salida esperada.

Otros sistemas [19, 75] cuentan con un enfoque de caja blanca: no sólo prueban el comportamiento de programas sueltos, sino que analizan la estructura del código fuente.

Soler et al. [115] presentan una herramienta de e-learning orientada a web para diagramas de clase UML. El entorno UML que proponen tiene la capacidad de corregir automáticamente

⁵<https://www.codingame.com/>

⁶<https://www.aceptaelreto.com/>

ejercicios de diagramas de clase UML y proporcionar un feedback a los estudiantes de forma inmediata. La herramienta forma parte de un entorno más general, llamado ACME [4], que proporciona las funcionalidades principales de una plataforma de e-learning. El estudiante dibuja un diagrama UML en la interfaz gráfica de la herramienta, que permite que se introduzcan diagramas hasta que se dé con la solución o se llegue al límite de tiempo. El sistema registra todos los intentos hasta que se obtiene la solución. Toda esta información resulta muy valiosa para el profesor, y puede utilizarse en la evaluación del alumno. Para cada ejercicio se almacena un conjunto de posibles soluciones, que se codifican utilizando un formato específico. Una vez el estudiante introduce su solución, el sistema inicia su corrección mediante un proceso de comparación. Se compara la solución introducida por el estudiante con todas las soluciones posibles del problema almacenadas en el repositorio del sistema. Si ninguna de las soluciones correctas coincide con la del estudiante, el sistema marca la solución del estudiante como incorrecta, y envía al estudiante un mensaje de feedback utilizando la solución que más se acerca a la que él ha enviado, orientándole de esta forma a llegar a la solución.

Benac et al. [17] proponen un sistema para calificar de forma automática ejercicios de programación utilizando pruebas basadas en propiedades. La calificación se lleva a cabo teniendo en cuenta la corrección funcional (errores), la complejidad del algoritmo, y la complejidad estructural. Para obtener una medida de la corrección funcional del programa se utiliza la herramienta QuickCheck [11]. QuickCheck genera una instanciación aleatoria de las variables obtenidas a partir de una propiedad booleana codificada en el dato, y comprueba que la propiedad booleana de la instanciación aleatoria es `true`. Este procedimiento se repite 100 veces, y en el caso de que la propiedad booleana de la instanciación sea `false`, o se produzca una excepción, se ha encontrado un error y la prueba termina.

Para establecer una puntuación de la calidad del algoritmo, Benac et al. proponen un procedimiento mediante el que se decide que el programa A es peor que el programa B si los fallos del programa B están incluidos en los fallos del programa A. Para calificar esta característica proporcionan además una herramienta⁷ que construye un árbol de calificación en el que los nodos son los programas que se ejecutan de forma idéntica y las ramas son las pruebas, de forma que todos los nodos debajo de una rama han fallado esas pruebas, y todos los nodos encima de una rama han ejecutado esas pruebas correctamente.

Para obtener una medida de la complejidad del algoritmo proporcionan una herramienta de “Complejidad”⁸ a partir de la que se obtiene una puntuación basada en la ejecución de peor rendimiento. A continuación construyen unas gráficas con las puntuaciones de todos los

⁷<https://github.com/fredlund/Ranker>

⁸<https://github.com/prowessproject/complexity>

programas e interpretan los resultados manualmente. La medida de la complejidad del código la obtienen utilizando la herramienta JavaNCSS⁹, mediante la que extraen dos métricas de los programas: la métrica *non-commenting source statements* (NCSS) y la métrica de complejidad ciclomática de McCabe. Se utilizan unas heurísticas para calificar los ejercicios en cuanto a la corrección funcional, la complejidad algorítmica, y la complejidad estructural.

Como se puede observar, todas estas herramientas son específicas para un dominio y difíciles de extender, cuestiones que la herramienta WODEL-EDU que presentamos en esta tesis pretende resolver (ver Sección 4).

2.4. Conclusiones del estudio

Nuestro estudio del estado del arte sobre técnicas de mutación nos permite concluir que los marcos y herramientas existentes para mutación de modelos son específicos para un lenguaje de programación o de modelado, o para un dominio de aplicación. Además, normalmente se utilizan lenguajes de propósito general que no están orientados a la definición y generación de mutantes. Estos enfoques requieren un mayor trabajo del programador, que debe enfrentarse a detalles accidentales de la plataforma de implementación, por ejemplo, para realizar la carga y serialización de modelos, comprobar si los mutantes generados son válidos, etc. Por tanto, existe una carencia de propuestas que faciliten la definición y prueba sistemática de operadores de mutación y que puedan aplicarse a diferentes lenguajes y aplicaciones. Por este motivo, esta tesis propone el DSL WODEL para la definición de operadores de mutación. WODEL proporciona: primitivas de alto nivel (p. ej., para la clonación o retipado de objetos) así como estrategias para su composición; integración con aplicaciones externas a través de la compilación a un lenguaje de propósito general; trazabilidad de las mutaciones aplicadas; y comprobación de mutantes duplicados. El Capítulo 3 describe en detalle el DSL WODEL.

La revisión de las herramientas de pruebas de mutación muestra que típicamente son específicas para un lenguaje (normalmente de programación) y dominio. Para superar esta restricción, esta tesis propone un enfoque nuevo con WODEL-TEST que facilita la creación de herramientas de pruebas de mutación para cualquier lenguaje de programación o de modelado definido por un meta-modelo. WODEL-TEST utiliza el DSL WODEL para definir los operadores de mutación para el lenguaje objetivo [50, 51] y utiliza MDE para generar la herramienta de pruebas de mutación personalizada para cada lenguaje. El Capítulo 5 presenta la herramienta WODEL-TEST.

Por otro lado, los entornos de generación automática de ejercicios revisados también son específicos para un dominio determinado, como ejercicios de programación [101], o

⁹<http://www.kclee.de/clemens/java/javancss/>

ejercicios de diagramas de clase UML [115]. Por tanto, nuestra propuesta del entorno `WODEL-EDU` para la generación automática de ejercicios independiente del dominio es un enfoque novedoso. Se describe la herramienta `WODEL-EDU` en el Capítulo 4.

Capítulo 3

Wodel: Un DSL para Mutación de Modelos

En este capítulo se presenta la solución propuesta en esta tesis para facilitar la creación de programas de mutación. Dicha solución se basa en la utilización de un DSL llamado `WODEL`, cuya sintaxis y funcionalidad se presenta en la Sección 3.1. Posteriormente, la Sección 3.2 describe las distintas funcionalidades proporcionadas por nuestra herramienta para el soporte al proceso de mutación: la configuración de la validación de los mutantes (Sección 3.2.1); el registro de los operadores de mutación aplicados (Sección 3.2.2); la detección de mutantes equivalentes (Sección 3.2.3); y la posibilidad de realizar acciones de post-procesado de los mutantes generados (Sección 3.2.4). A continuación, la Sección 3.3 presenta el soporte que proporciona nuestra herramienta para la definición de programas de mutación: la síntesis de modelos semilla a partir de las instrucciones incluidas en el programa `WODEL` (Sección 3.3.1); y las métricas de mutación (Sección 3.3.2). La siguiente Sección 3.4 muestra la herramienta proporcionada, y por último, en la Sección 3.5 se presenta una evaluación del DSL `WODEL` que consiste en la comparación del esfuerzo requerido para implementar un programa de mutación de modelos en nuestro DSL con el que requeriría implementar este mismo programa en Java.

3.1. El DSL `WODEL`

Para facilitar la especificación y la creación de mutación de modelos de forma independiente del meta-modelo, proponemos el DSL `WODEL`. Este DSL proporciona primitivas para mutación de modelos (p. ej., creación, borrado, redirección de referencias), estrategias de selección de elementos (p. ej., aleatoria, específica, todos), y composición de operadores

- *Creación de objetos.* La clase `CreateObject` crea un objeto de la clase indicada por la referencia `type` e inicializa sus campos. De forma opcional, es posible seleccionar un objeto contenedor para el objeto creado utilizando una estrategia `ObjectSelectionStrategy`. En ese caso, la referencia `refType` indica la referencia contenedora donde se ubicará el nuevo objeto. La sintaxis concreta de este operador de mutación queda definida por la primitiva `create`. El nuevo objeto se coloca dentro de la referencia de composición especificada de un objeto dado, o si no se ha especificado una referencia contenedora, `WODEL` selecciona un contenedor compatible de forma aleatoria. Los atributos y referencias obligatorios se inicializan de forma automática a un valor aleatorio si no se les asigna uno de forma explícita en la instrucción `WODEL`. Esta primitiva se utiliza en la Figura 3.2 para crear un objeto de tipo `Transition` y asignarle un objeto de tipo `Symbol` seleccionado de forma aleatoria. El ejemplo muestra además la inicialización automática de las referencias `src` y `tar` de la transición creada.

```
// crea una transicion y le asigna un simbolo aleatorio
create Transition with {symbol = one Symbol}
```

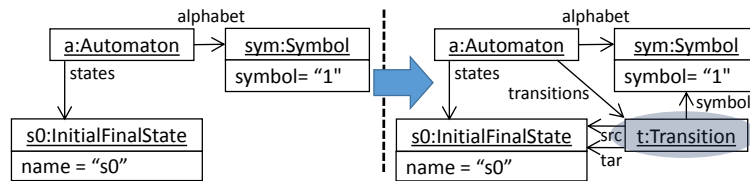


Figura 3.2 Ejemplo de operador de mutación de creación de objetos.

- *Clonación de objetos.* La clase `CloneObject` (no incluida en el meta-modelo de la Figura 3.1) realiza una copia de un objeto. Se puede indicar que dicha copia sea superficial (sin incluir en la copia los objetos alcanzables desde el objeto clonado mediante referencias de composición) o completa (haciendo también una copia de todos los objetos alcanzables desde el objeto clonado mediante referencias de composición). La sintaxis concreta de este operador de mutación viene definida por la primitiva `clone` a la que se añade la directiva `deep` en caso de querer realizar la copia completa. Como en la primitiva para creación de objetos, es posible especificar una referencia de composición para el clonado, de otra forma el clon se almacena en una referencia contenedora compatible aleatoria. La primitiva se ilustra en la Figura 3.3, donde un objeto `Transition` con símbolo “1” se clona en otro objeto `Transition` con símbolo “0”.

```
// clona un objeto transition con simbolo '1'
// y le asigna el simbolo '0'
sym0 = select one Symbol where {symbol = '0'}
sym1 = select one Symbol where {symbol = '1'}
clone one Transition where {symbol = sym1} with {symbol = sym0}
```

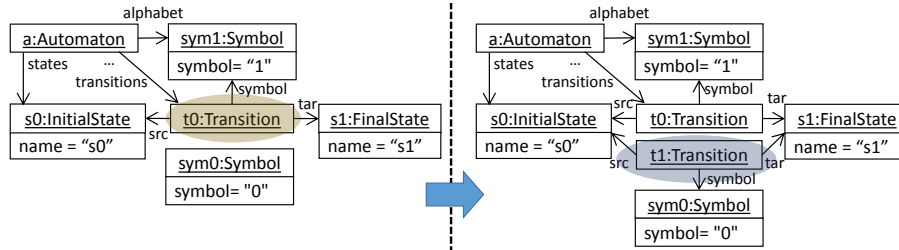


Figura 3.3 Ejemplo de operador de mutación de clonación de objetos.

- *Modificación de campos de un objeto.* La clase `ModifyInformation` selecciona un objeto mediante una `ObjectSelectionStrategy`, y proporciona un conjunto de modificaciones a realizar sobre sus atributos o referencias (clases `AttributeSet` y `ReferenceSet`). El meta-modelo muestra sólo algunas de las posibles modificaciones, como inicializar el valor de un atributo, intercambiar el valor de dos atributos o referencias, y copiar el valor de un atributo en otro. La sintaxis concreta de este operador de mutación viene definida por la primitiva `modify`. A modo de ejemplo, la Figura 3.4 muestra la modificación de referencia, y la Figura 3.5 la modificación de un atributo. En el primer caso, el programa `WODEL` modifica la referencia `tar` de una transición asignándole un objeto `State` diferente. En el segundo caso, el programa cambia el atributo `name` a un `State` de “q0” a “s0”.

Esta primitiva también puede utilizarse con los modificadores `source` y `target` para redirigir el origen o el destino de una referencia a otro objeto. Este caso está soportado por las clases `ModifySourceReference` y `ModifyTargetReference` del meta-modelo de `WODEL` (ver Figura 3.1). La Figura 3.6 muestra un ejemplo donde el símbolo de una transición se modifica por otro diferente.

```
// modifica el destino de una transicion por otro estado
t1 = select one Transition
s0 = select one State where {self <> t1->tar}
modify t1 with {tar = s0}
```

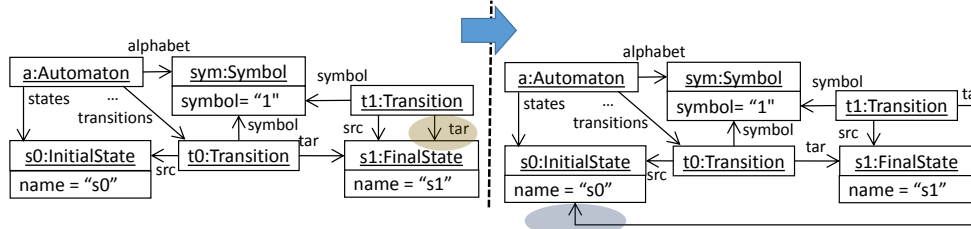


Figura 3.4 Ejemplo de operador de mutación de modificación de referencia.

```
// modifica el nombre de un estado
modify one State where {name = 's0'} with {name = 'q0'}
```

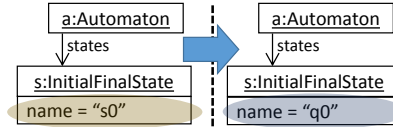


Figura 3.5 Ejemplo de operador de mutación de modificación de atributo.

```
// modifica el simbolo de una transicion por otro diferente
modify target symbol from one Transition to one Symbol
```

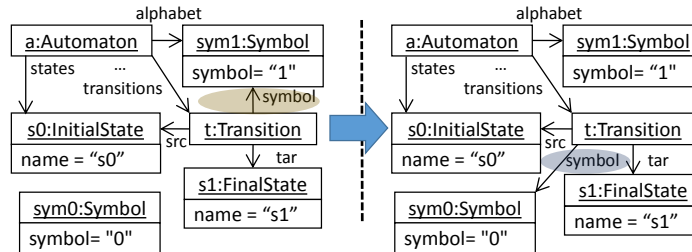


Figura 3.6 Ejemplo de operador de mutación de modificación de destino de referencia.

- *Retipado de objetos.* La clase `RetypeObject` se utiliza para retipar un objeto a otro de sus tipos hermanos, conservando todos los atributos y referencias compatibles del objeto original. La sintaxis concreta de este operador de mutación queda definida por la primitiva `retype`. En el ejemplo de la Figura 3.7, el operador retipa un objeto de tipo `InitialState` a un objeto `InitialFinalState`.

```
// retipa un estado inicial a un estado inicial y final
retype one InitialState as InitialFinalState
```

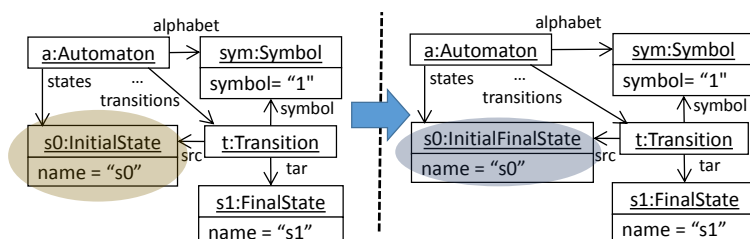


Figura 3.7 Ejemplo de operador de mutación de retipado de objetos.

- *Borrado de objetos.* La clase `RemoveObject` borra de forma segura un objeto seleccionado por una estrategia `ObjectSelectionStrategy`, asegurando que no queda vacía ninguna referencia obligatoria origen/destino. La sintaxis concreta de este operador de mutación viene definida por la primitiva `remove`. `WODEL` proporciona además la primitiva `remove reference` (clase `RemoveReferenceMutation` del meta-modelo de `WODEL`) que borra una referencia del tipo dado entre dos objetos seleccionados. Se muestra un ejemplo del borrado de objetos en la Figura 3.8, donde el programa `WODEL` borra el objeto `Symbol` de una transición.

```
// borra el simbolo de una transicion
remove one Symbol
```

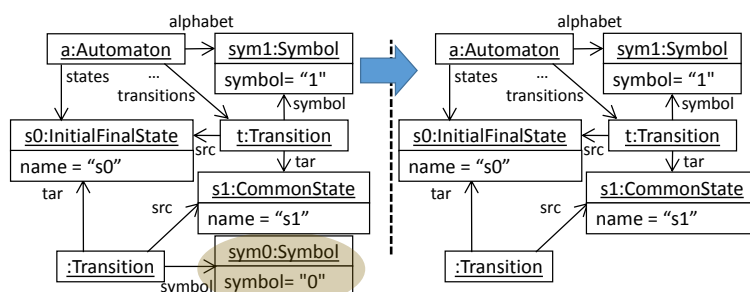


Figura 3.8 Ejemplo de operador de mutación de borrado de objetos.

Un programa `WODEL` tiene dos partes. La primera declara el número de mutantes a generar, o todos los mutantes posibles para cada operador de mutación mediante la palabra clave `exhaustive`; la carpeta de salida; y la carpeta donde se encuentran los modelos semilla y su meta-modelo. La segunda parte define los operadores de mutación y cuántas veces deben aplicarse. Opcionalmente, los programas pueden incluir una lista de restricciones OCL que todos los mutantes generados deben cumplir.

A continuación se muestran varios programas `WODEL` que ilustran las distintas características de este DSL.

El Listado 3.1 muestra un programa `WODEL` sencillo. La línea 1 establece que se quieren generar 3 mutantes en la carpeta `out`, a partir de los modelos semilla incluidos en la carpeta `models`. La línea 2 indica que en este ejemplo se utiliza el meta-modelo de la Figura 2.2, que debe especificarse de manera externa al programa `WODEL` (ver Sección 3.4). La línea 3 proporciona una descripción del programa, que es opcional. Las líneas 6-11 del Listado 3.1 contienen el cuerpo del programa con los operadores de mutación. Los operadores pueden organizarse en bloques (línea 6) y utilizar los mutantes generados por otros bloques como modelos semilla, p. ej., para definir operadores de mutación de orden superior. Las líneas 8-10 definen las tres instrucciones que constituyen la implementación del operador de mutación llamado `CNFS`. La primera (línea 8) selecciona de forma aleatoria un estado final, y lo convierte en no final; la segunda (línea 9) crea un nuevo estado final; y la última (línea 10) crea una nueva transición desde el estado modificado en la línea 8 al creado en la línea 9, asignándole un símbolo aleatorio del alfabeto.

Todos los tipos de operadores de mutación heredan de la clase `Mutation`, que contiene el número mínimo y máximo de veces que el operador de mutación se va a aplicar. Si se omite la información, como en el operador de mutación del Listado 3.1, el operador se aplica una única vez. A su vez, `Mutation` hereda de `ObjectEmitter`, y por tanto puede recibir un nombre, de forma que se puede referenciar desde otros operadores de mutación. Por ejemplo, en la línea 10 del Listado 3.1, el nombre `s0` se utiliza para referirse al objeto `FinalState` retipado en la línea 8.

```

1 generate 3 mutants in "out/" from "models/"
2 metamodel "http://fa.com"
3 description "Simple Wodel program"
4
5 with blocks {
6   CNFS "Changes a final state to non-final and connects it with a new final state"
7   {
8     s0 = retype one FinalState as CommonState
9     s1 = create FinalState
10    t0 = create Transition with {src = s0, tar = s1, symbol = one Symbol}
11  }
12 }
```

Listado 3.1 Un programa `WODEL` sencillo.

Los objetos y las referencias utilizados en los operadores de mutación pueden seleccionarse utilizando las siguientes estrategias: selección aleatoria, específica (referenciada mediante el nombre asignado al objeto), basada en alguna propiedad, todos los objetos que son de un tipo dado, y un objeto diferente al seleccionado en el operador de mutación actual. El meta-modelo de la Figura 3.9 muestra la clase abstracta `ObjectSelectionStrategy` que define las estrategias de selección, y muestra tres de estas estrategias. `SpecificObjectSelection` selecciona un objeto referenciado por un emisor. `SpecificReferenceSelection` seleccio-

na a la vez un objeto y una referencia definida por la clase del objeto. `RandomObjectSelection` selecciona un objeto aleatorio que es una instancia de la clase especificada por la referencia `type`. Todas las estrategias pueden configurarse con una condición (de la clase `Expression`) sobre los valores de los atributos y referencias de los elementos seleccionados. Por ejemplo, la instrucción de mutación de modificación de atributo en la Figura 3.5 utiliza una selección de estrategia aleatoria (`one State`) cuyo parámetro es una condición de atributo (`where {name = 'q0'}`).

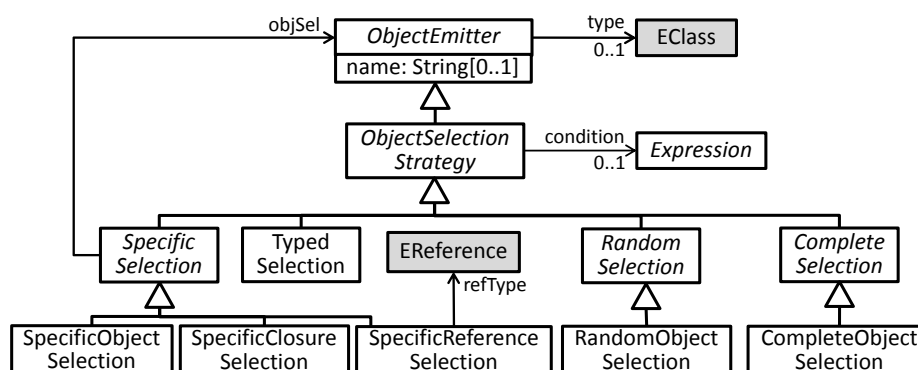


Figura 3.9 Estrategias de selección soportadas.

Como muestra el Listado 3.1, `WODEL` tiene una sintaxis concreta textual. Una sintaxis gráfica - quizá similar a reglas de transformación de grafos [41] - se podría haber elegido en su lugar para expresar los operadores de mutación. Sin embargo, se ha optado por una sintaxis textual debido a su concisión, la necesidad de incluir expresiones para mutación de atributos y referencias, facilitar la referencia de operadores de mutación desde otros, y permitir su agregación en mutaciones compuestas.

El Listado 3.2 muestra una parte pequeña de la gramática de `WODEL` (la gramática completa se incluye en el Anexo A). La regla `DEFINITION` declara el meta-modelo que se utiliza, el número de mutantes que se generarán o todos los mutantes posibles en el modo exhaustivo, la ubicación de los modelos semilla, la definición opcional de recursos adicionales sobre los que se podrán realizar consultas, y la descripción opcional del objetivo del programa para facilitar su identificación. A esta definición le sigue una secuencia de instrucciones de mutación (`MUTATION*`), que pueden ejecutarse secuencialmente o embebidas en bloques (`BLOCK*`). Este listado muestra además las reglas para las primitivas de mutación `CREATEOBJECT`, `MODIFYINFORMATION` y `MODIFYSOURCEREFERENCE`. Cada una de estas primitivas de mutación se reduce en la regla `MUTATION` donde se permite incluir la cardinalidad `min` y `max` que indica el número de veces que dicha primitiva se aplicará. Por último, la regla `COMPOSITEMUTATION` permite la declaración de una mutación compuesta de varias instrucciones `MUTATION` que se aplicarán en un único paso.

```

1 WODELPROGRAM ::= DEFINITION
2   with ( commands { MUTATION* } | blocks { BLOCK* } )
3   ( constraints { CONSTRAINT* } )?
4
5 DEFINITION ::=
6   generate (<num>|exhaustive) mutants in <folder> from SEEDS
7   metamodel <meta-model>
8   ( with resources from { RESOURCE } ( , { RESOURCE } ) * )?
9   ( description <description> )?
10
11 MUTATION ::=
12   ( CREATEOBJECT | MODIFYINFORMATION |
13     MODIFYSOURCEREERENCE | ... | COMPOSITEMUTATION )
14   ( [ (<min> ..)? <max> ] )?
15
16 CREATEOBJECT ::=
17   ( <name> '=' )? create <EClass>
18   ( in OBJECTSELECTIONSTRATEGY ( '.' <EReference> )? )?
19   ( with { FEATURESET ( , FEATURESET ) * } )?
20
21 MODIFYINFORMATION ::=
22   ( <name> '=' )? modify OBJECTSELECTIONSTRATEGY
23   with { FEATURESET ( , FEATURESET ) * }
24
25 MODIFYSOURCEREERENCE ::=
26   modify source <EReference>
27   ( from OBJECTSELECTIONSTRATEGY )?
28   ( to OBJECTSELECTIONSTRATEGY )?
29
30 COMPOSITEMUTATION ::= ( <name> '=' )? [ MUTATION* ]
31 ...

```

Listado 3.2 Parte de la gramática de WODEL.

El Listado 3.3 muestra un nuevo ejemplo de un programa WODEL. Genera 3 mutantes a partir de todos los modelos semilla en la carpeta `models` (línea 1). Cada mutante se obtiene aplicando los operadores de mutación especificados un número aleatorio de veces de acuerdo con el intervalo de la cardinalidad del operador de mutación. Las líneas 5-7 definen un operador de mutación compuesto que elimina un estado no inicial aleatorio (línea 5), así como todos los objetos `Transition` a los que apunta el estado eliminado o que parten de él (líneas 6-7). En concreto, las transiciones que se eliminan son aquellas que tienen un valor indefinido en las referencias `src` o `tar`, que son las que apuntan al estado borrado. El operador de mutación en las líneas 8-10 selecciona un objeto `Transition` arbitrario, y modifica su referencia `symbol` para que apunte a un objeto `Symbol` diferente. Ambos operadores de mutación declaran un intervalo de cardinalidad que controla el número de aplicaciones del operador en cada mutante: el primer operador de mutación se aplicará un número aleatorio de veces con probabilidad uniforme entre 0 y 2, y el segundo entre 1 y 3.

```

1 generate 3 mutants in "out/" from "models/"
2 metamodel "http://fa.com"

```

```

3
4 with commands {
5   c0 = [ remove one State where {self not typed InitialState}
6         remove all Transition where {src = null}
7         remove all Transition where {tar = null} ] [0..2]
8   modify target symbol
9         from one Transition
10        to other Symbol [1..3]
11 }

```

Listado 3.3 Operadores de mutación compuestos y cardinalidades.

En el Listado 3.4, todos los operadores de mutación se ejecutan secuencialmente sobre los modelos semilla especificados en la cabecera del programa. Además, `WODEL` permite organizar las instrucciones de mutación en bloques con nombre y una descripción opcional, que toman como semilla o bien los mutantes generados por algunos bloques previos seleccionados, o bien los modelos semilla indicados en la cabecera del programa si no se ha indicado otro bloque. El Listado 3.4 muestra un ejemplo de un programa `WODEL` que utiliza mutantes generados en bloques previos. Se declaran dos bloques, uno llamado `first` que genera 2 mutantes a partir del modelo semilla `evenBinary.fa` indicado en la cabecera del programa (líneas 5-7), y otro llamado `second` que genera 3 mutantes a partir de cada mutante producido por el bloque `first` (líneas 8-10). La directiva `repeat=no` en el segundo bloque asegura que los mutantes que produce el bloque `second` sean distintos del modelo semilla. En total, en este ejemplo se generarían como máximo 8 mutantes (2 correspondientes al primer bloque, y en el segundo bloque, 3 mutantes por cada uno de los mutantes generados en el primer bloque). Mediante esta estructura de instrucciones `WODEL` en bloques encadenados se puede construir un árbol de mutantes. Esta estructura de árbol de mutantes resulta de utilidad, p. ej., en la generación automática de ejercicios (ver Capítulo 4). También, la utilización de los bloques nos permite aislar instrucciones `WODEL` con el objetivo de utilizar cada bloque como un operador de mutación, cuestión que resulta de utilidad en la definición de operadores de mutación de la herramienta de síntesis de herramientas de MT (ver Capítulo 5).

```

1 generate mutants in "out/" from "evenBinary.fa"
2 metamodel "http://fa.com"
3
4 with blocks {
5   first {
6     remove one Transition
7   } [2]
8   second from first repeat=no {
9     create Transition
10  } [3]
11 }

```

Listado 3.4 Bloques de instrucciones de mutación.

WODEL asegura que los mutantes que se generan son válidos, esto es, conformes al meta-modelo indicado en la cabecera. Sin embargo, en ocasiones puedes querer que los mutantes cumplan restricciones adicionales no incluidas en el meta-modelo. Para ello WODEL permite definir restricciones OCL adicionales que todos los mutantes generados deben cumplir. A modo de ejemplo, el Listado 3.5 muestra un programa WODEL en el que se incluye una restricción OCL que todos los mutantes generados deben cumplir. El bloque `simple` define un operador de mutación que selecciona un estado inicial `s0` (línea 6), un estado no final `s1` (línea 7), una transición `t0` cuyo estado origen es el estado inicial `s0` seleccionado (línea 8), e intercambia el estado destino de esta transición con el de otra transición cuyo estado destino es el estado no final `s1` seleccionado (línea 9). La restricción OCL definida en las líneas 13-15 obliga a que todos los estados de los mutantes generados sean alcanzables desde el estado inicial. Estas restricciones OCL adicionales se comprueban una vez se genera cada mutante. El mutante queda descartado si alguna de estas restricciones no se cumple, repitiéndose el proceso hasta que se obtiene un mutante que satisface todas las restricciones o se alcanza un número predefinido de intentos.

```

1 generate 2 mutants in "out/" from "model/"
2 metamodel "http://fa.com"
3
4 with blocks {
5   simple {
6     s0 = select one InitialState
7     s1 = select one State where {self not typed FinalState}
8     t0 = select one Transition where {src = s0}
9     modify one Transition where {tar = s1} with {swap(tar, t0→tar)}
10  } [2]
11 }
12 constraints {
13   context State connected:
14     "oclIsKindOf(InitialState) or
15     Set{self}→closure(s |
16       Transition.allInstances()→select(t | t.tar=s)→collect(src))
17     →exists(s | s.oclIsKindOf(InitialState))"
18 }

```

Listado 3.5 Definición de restricciones OCL.

El ejemplo del Listado 3.6 presenta un programa WODEL para autómatas en el que se utilizan recursos adicionales, es decir, otros modelos definidos por un meta-modelo, sobre los que se puede realizar consultas. La capacidad de realizar consultas a recursos adicionales dentro del programa WODEL puede ser utilizada para, por ejemplo, la implementación de operadores de mutación que pueden necesitar realizar consultas a símbolos de un alfabeto distinto al incluido en el modelo de autómata finito que se está mutando. En el ejemplo del Listado 3.6, la declaración de los recursos adicionales se indica en la línea 3. En concreto, hay que proporcionar la ubicación de los recursos adicionales y su meta-modelo. En este ejemplo

sólo se utiliza un recurso adicional, pero se pueden añadir tantos como sea necesario, separándolos en la declaración mediante comas. Mediante esta directiva se indica que el programa `WODEL` podrá realizar consultas sobre los recursos de símbolos del alfabeto ubicados en la carpeta `model/alphabet` y que son conformes al meta-modelo `http://alphabet.com`. Las líneas 6-9 definen un operador de mutación que cambia uno de los símbolos del alfabeto del autómatas semilla por un símbolo diferente extraído del recurso adicional.

```
1 generate 2 mutants in "out/" from "model/"
2 metamodel "http://fa.com"
3 with resources from {alphabet="model/alphabet" metamodel="http://alphabet.com"}
4
5 with blocks {
6   ms {
7     sym = select one Symbol from alphabet
8     modify one Symbol where {symbol <> sym} with {symbol = sym}
9   }
10 }
```

Listado 3.6 Utilización de recursos adicionales.

Si bien se puede utilizar un lenguaje de programación de propósito general como Java para implementar los operadores de mutación de modelos, el DSL `WODEL` proporciona facilidades específicas para mutación de modelos. Por ejemplo, selecciona de forma automática un contenedor apropiado cuando se crean objetos nuevos o se clonan objetos, asegura que no queda ninguna referencia de origen/destino a objetos borrados, y da soporte a la definición de restricciones OCL adicionales que todos los mutantes generados deben satisfacer.

Los programas `WODEL` correctos se compilan automáticamente a código Java. El código Java generado se encarga de crear los mutantes a partir de los modelos semilla. La ventaja de generar código Java de forma explícita es que puede utilizarse en aplicaciones independientes. Además, este código es genérico dado que maneja modelos utilizando la API reflexiva de EMF [118], y así, puede reutilizarse para mutar cualquier modelo que sea conforme al meta-modelo de dominio.

Una vez vista la sintaxis de `WODEL`, en la siguiente sección presentamos otras utilidades complementarias al lenguaje, que se han desarrollado para facilitar la creación de programas de mutación y aplicaciones basadas en mutación.

3.2. Soporte al proceso de mutación

`WODEL` proporciona funcionalidades avanzadas para el proceso de mutación como la validación de los mutantes generados, un registro de las mutaciones aplicadas, criterios para la detección de equivalencias sintácticas y semánticas de los mutantes, y facilidades para

utilizar los mutantes generados en aplicaciones de post-procesado. Estas funcionalidades se presentan en las siguientes subsecciones.

3.2.1. Validación de mutantes

Como se ha mencionado en la Sección 3.1, `WODEL` comprueba que los mutantes generados son conformes al meta-modelo del dominio y satisfacen sus restricciones de integridad. Esta validación de modelos se activa de forma opcional. Los mutantes no válidos quedan descartados, y el motor intenta generar otro mutante hasta un número máximo de intentos que es configurable. Además, los programas `WODEL` pueden incluir restricciones OCL [94] que cualquier mutante generado debe satisfacer, como muestra el Listado 3.5. Estas restricciones también se validan para cada mutante generado, descartándolo si no se cumplen.

Sin embargo, algunos dominios pueden requerir que los modelos generados tengan ciertas propiedades que no son fácilmente expresables con OCL. Por ejemplo, en el dominio de los autómatas finitos puede requerirse que el lenguaje definido por uno de los autómatas sea el mismo que el lenguaje que define la concatenación de otros dos. Para tratar estos casos, `WODEL` puede extenderse con nuevos criterios de validación de modelos implementados en Java. Como veremos en la Sección 3.4, que presenta la herramienta de soporte, eso es posible mediante el uso de puntos de extensión.

3.2.2. Registro de mutaciones

A veces, las aplicaciones basadas en mutación necesitan una trazabilidad entre los modelos semilla y los mutantes generados para saber qué elementos se han mutado y en qué orden, o tener conocimiento de los operadores de mutación utilizados para generar los mutantes. Por ejemplo, la Sección 4.3 presenta una aplicación donde la solución correcta a un ejercicio se muta para obtener soluciones incorrectas, y se requiere acceder a la lista de operadores de mutación utilizados con el objetivo de sintetizar frases que expliquen cómo corregir la solución incorrecta mediante la aplicación inversa de los operadores de mutación usados.

Por tanto, cuando se ejecuta un programa `WODEL` es posible generar un registro con la secuencia de los operadores de mutación aplicados y el contexto de su aplicación para cada mutante generado. Los registros de operadores de mutación aplicados son a su vez modelos, lo que facilita que sean consultados o manipulados posteriormente.

La Figura 3.10 muestra una parte del meta-modelo del registro de operadores de mutación aplicados. La clase `Registry` contiene una lista ordenada de objetos de tipo `AppliedMutation`, cada uno de los cuales almacena una referencia a la instrucción `Mutation` ejecutada en el

programa `WODEL` y utilizada para generar el mutante (ver Figura 3.1). Esto es posible porque los programas `WODEL` son modelos a su vez (conformes al meta-modelo de `WODEL` mostrado en las Figuras 3.1 y 3.9). La clase `AppliedMutation` tiene una subclase por cada subclase de `Mutation`, es decir, por cada posible operador de mutación. De este modo, dependiendo de la instrucción particular ejecutada, se instancia la subclase adecuada de `AppliedMutation`. Por ejemplo, para cada ejecución del operador de mutación `CreateObject` en un programa, se añade un objeto `ObjectCreated` al modelo del registro que almacena la referencia a la instrucción y al objeto creado. Si bien algunos de los objetos `AppliedMutation` necesitan referirse a elementos en el modelo semilla (p. ej., los objetos eliminados del objeto semilla no están presentes en el mutante), otros necesitan referirse al mutante resultante (p. ej., los objetos creados como resultado de aplicar los operadores de mutación no pertenecen al modelo semilla). En el caso de la clase `InformationChanged`, la referencia `object` indica el objeto del modelo que se ha mutado, y las referencias `refs` y `atts` proporcionan respectivamente la información que corresponde a las referencias y atributos de ese objeto que se han modificado. En cuanto a la clase `ReferenceChanged`, la referencia `object` indica el objeto sobre el que se ha aplicado la mutación.

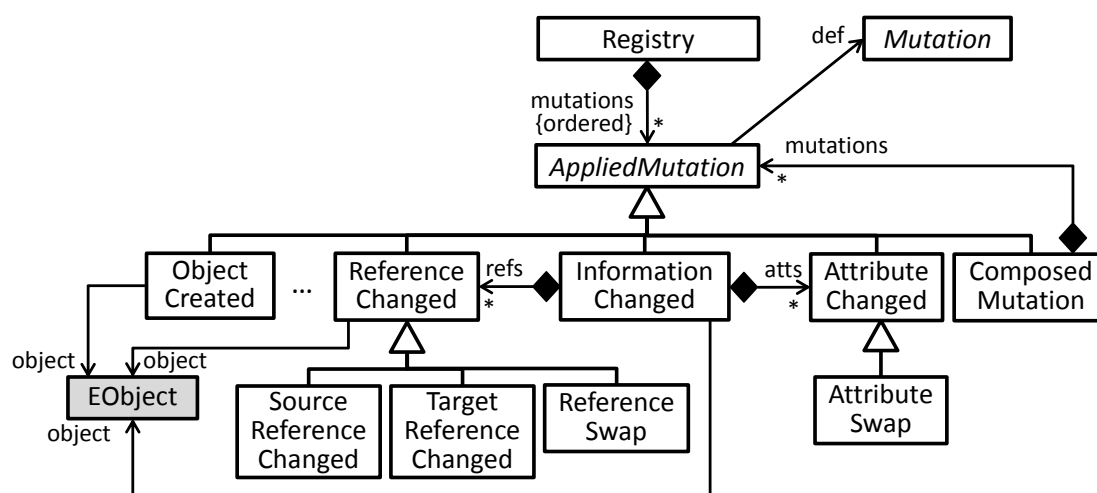


Figura 3.10 Meta-modelo del registro de los operadores de mutación (fragmento).

El registro guarda *todas* las mutaciones aplicadas. Esto significa que incluso si el efecto de la aplicación de un operador de mutación se deshace por un operador subsecuente (p. ej., el valor de un atributo booleano es invertido dos veces), se guardan ambas mutaciones en el registro. Dado que algunas aplicaciones pueden considerar irrelevantes los operadores de mutación que finalmente se revierten, es posible eliminar del registro de operadores de mutación aplicados aquellos que se cancelan entre sí, obteniendo por tanto una descripción más compacta de todas las diferencias entre el modelo semilla y el mutante generado. Nótese

que esto proporciona más información que una simple comparación de los modelos semilla y mutante, ya que se tiene información no sólo de lo que ha cambiado, sino también mediante qué operador. Las compactaciones del registro incluidas son las siguientes:

- Creación de un objeto y eliminación posterior.
- Modificación del valor de un atributo y posterior modificación al valor previo del atributo.
- Intercambio de los valores de dos atributos que se vuelven a intercambiar posteriormente.
- Modificación del valor de una referencia y posterior modificación al valor previo de la referencia.
- Intercambio de los valores de dos referencias que se vuelven a intercambiar posteriormente.

El registro de operadores de mutación aplicados puede utilizarse para replicar el proceso de mutación, o para propósitos específicos de la aplicación. Por ejemplo, el registro de operadores de mutación se ha utilizado para sintetizar a lenguaje natural la descripción de los operadores de mutación, con el objetivo de incluir estas descripciones en la generación automática de ejercicios [50], o también en las pruebas de mutación, para la inclusión de comentarios en el código de los programas mutados que identifican el operador de mutación que se ha aplicado sobre el programa [52, 53]. Estas aplicaciones se presentarán en los capítulos 4 y 5, respectivamente.

3.2.3. Detección de mutantes equivalentes

Asegurar la unicidad de los mutantes es útil en algunas aplicaciones, como la generación automática de ejercicios [50], o las pruebas de mutación [52, 53]. Se distingue la equivalencia sintáctica de la equivalencia semántica de los mutantes generados con respecto al modelo semilla a partir del que se crean. La equivalencia sintáctica se refiere a mutantes duplicados o que tienen una sintaxis equivalente. Por ejemplo, en la mutación de programas Java, un mutante donde se ha borrado un comentario sería sintácticamente equivalente al programa original. La equivalencia semántica incluye comprobación en las semánticas del modelo o de su compilación (p. ej. considerando que dos mutantes Java son equivalentes si se compilan al mismo bytecode). Por ejemplo, en la mutación de programas Java, un operador de mutación que añade “+0” a una expresión aritmética da como resultado un mutante semánticamente

equivalente, y el mutante es diferente sintácticamente al programa original. Esta distinción es relevante para las pruebas de mutación dado que los mutantes equivalentes se comportan de igual forma que el programa original; por tanto, no añaden ninguna información y pueden eliminarse para mejorar la eficiencia del proceso de pruebas de mutación.

Para facilitar la definición y detección de mutantes equivalentes, `WODEL` incluye un mecanismo de extensión para definir criterios de equivalencia sintáctica y semántica. Estos criterios se evalúan en cada mutante generado. Si se detecta un mutante equivalente sintácticamente, se borra y se procede a generar el mutante siguiente. En caso de detectar un mutante equivalente semánticamente, se marca como tal y se excluye del proceso de pruebas de mutación.

3.2.4. Post-procesado de mutantes

La herramienta `WODEL` proporciona un punto de extensión para post-procesado de mutantes que resulta útil para facilitar que cualquier aplicación externa pueda utilizar los mutantes generados por `WODEL`. Las aplicaciones externas pueden proporcionar el código que realiza la acción de post-procesado de los mutantes en este punto de extensión de manera no intrusiva. En la Sección 3.4 se darán detalles más concretos sobre la implementación de esta funcionalidad.

3.3. Soporte a la definición de programas de mutación

`WODEL` proporciona además funcionalidades para la ingeniería de operadores de mutación. Estas funcionalidades incluyen la síntesis de modelos semilla para probar los operadores, métricas que analizan la cobertura de los operadores, y servicios para generar operadores en el caso de que la cobertura sea insuficiente. Estas funcionalidades se describen en las siguientes subsecciones.

3.3.1. Síntesis de modelos semilla

En esta Subsección se presenta la síntesis de modelos semilla. La utilidad de estos modelos es facilitar la realización de pruebas destinadas a comprobar la correcta implementación de los operadores de mutación. Dado un programa `WODEL`, el proceso de síntesis genera un modelo semilla sobre el que todas las instrucciones del programa son aplicables (si tal modelo existe dentro del ámbito de búsqueda definido). De este modo, el desarrollador puede ejecutar el programa `WODEL` con los modelos semilla generados y comprobar si obtiene el resultado esperado.

La Figura 3.11 describe el proceso de generación de modelos semilla. Ese proceso se basa en la búsqueda de modelos, una técnica que aplica resolución de restricciones sobre los modelos [62]. En concreto, el sintetizador recibe como entrada un programa `WODEL` y el meta-modelo de dominio (y sus invariantes) para el que está definido el programa. El sintetizador extiende el meta-modelo de dominio con restricciones OCL adicionales derivadas del programa `WODEL`. Estas restricciones expresan los requisitos que un modelo semilla debe cumplir para permitir la aplicación de cada operador de mutación incluido en el programa. A continuación, el meta-modelo enriquecido se carga en un buscador de modelos [74], que realiza una búsqueda acotada de las instancias del meta-modelo que satisfacen las restricciones OCL derivadas del programa `WODEL`. Si se encuentra un modelo, entonces asegura una cobertura completa de las instrucciones del programa `WODEL` al ejecutarlo sobre el mismo.

El ámbito de esta búsqueda acotada está formado por modelos con un número máximo de objetos de cada tipo. Por ejemplo, en un ámbito con un número máximo de cinco elementos se incluyen modelos que tienen como mucho cinco elementos de cada tipo. El razonamiento que se sigue se denomina “la hipótesis del ámbito pequeño”, que afirma que la mayoría de las violaciones de las restricciones se pueden descubrir con contraejemplos pequeños. Utilizando un ámbito de búsqueda acotado, el espacio de búsqueda sigue siendo bastante grande. Por tanto, no se realiza una búsqueda explícita, sino que se traduce el problema a un “problema de satisfacibilidad” en el que las variables en lugar de ser relaciones, son bits. Cambiando individualmente el valor de los bits, un “*satisfiability (SAT) solver*” puede encontrar una solución. Si no se encuentra un modelo significa que no existe un modelo con las características requeridas en el espacio de búsqueda. Se puede encontrar un modelo ampliando el espacio de búsqueda, o no, ya que que el método no permite saberlo.

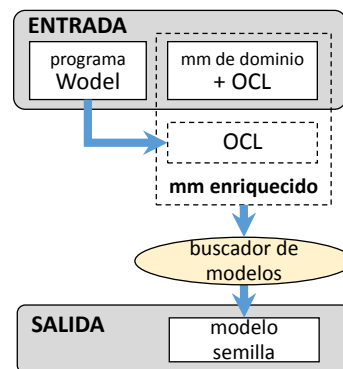


Figura 3.11 Generación de modelos semilla para un programa `WODEL`.

La Tabla 3.1 muestra las plantillas utilizadas para generar las restricciones OCL para cada primitiva de mutación (segunda columna), una explicación textual de la semántica de cada

Condiciones a comprobar	Plantilla OCL	Ejemplo
Filtrado de objetos: Plantilla auxiliar utilizada para comprobar que las características de un objeto tienen un valor dado.	<code>o.(feat₁) = <val₁> ... and o.(feat_n) = <val_n></code>	
Selección de objetos, modificación de objetos: Existe un objeto con el tipo y los valores de características dados.	<code><Class>.allInstances() →exists(o <object-filter>))</code>	Wodel: modify one State where {name = 'q0'} with {name = 's0'} OCL: <code>State.allInstances() →exists(s s.name = 'q0')</code>
Creación de objetos: Existe una referencia contenedora del tipo del objeto con espacio para añadir más objetos.	<code><Container>.allInstances() →exists(o o.(ref)→size() < <upB>))</code>	Wodel: a = create CommonState OCL: <code>Automaton.allInstances() →exists(a a.states→size() < 10)</code>
Eliminación de objetos: Existe un objeto con tipo y los valores de características dados, y su eliminación no viola la cardinalidad inferior de ninguna referencia del tipo del objeto.	<code><Class>.allInstances() →exists(o <object-filter> and <Container>.allInstances() →forAll(c c.(ref)→includes(o) implies c.(ref)→size() > <lowB>)))</code>	Wodel: remove one Symbol OCL: <code>Symbol.allInstances()→exists(s Automaton.allInstances() →forAll(a a.alphabet→includes(s) implies a.alphabet→size() > 1))</code>
Clonado de objetos: Existe un objeto con tipo y los valores de características dados, y existe una referencia de contención de ese tipo con espacio para añadir más objetos.	<code><Class>.allInstances() →exists(o <object-filter>)) and <Container>.allInstances() →exists(o o.(ref)→size() < <upB>))</code>	Wodel: clone one Transition OCL: <code>Transition.allInstances() →exists(t true) and Automaton.allInstances() →exists(a a.transitions→size() < 10)</code>
Retipado de objetos: Existe un objeto con el tipo de origen y los valores de características dados. Si el tipo de destino no es compatible con el contenedor del tipo de origen, se requieren condiciones adicionales para eliminar un objeto del tipo origen y crear uno del tipo destino (y lo mismo para las referencias incompatibles con el tipo destino). Concatenación or para cada tipo origen/destino considerados.	<code><Class>.allInstances() →exists(o <object-filter> [and <SrcContainer>.allInstances() →forAll(c c.(ref)→includes(o) implies c.(ref)→size() > <lowB>)) and <TrgContainer>.allInstances() →exists(c c.(ref)→size() < <upB>)]¹)</code> ¹ incluye la condición si <SrcContainer>.(ref) no es compatible con el tipo destino	Wodel: retype one CommonState as FinalState OCL: <code>CommonState.allInstances() →exists(s true)</code>
Creación de referencias: Existe un objeto del tipo de la referencia, y una referencia en la que se puede añadir el objeto sin violar la cardinalidad superior.	<code><TgtClass>.allInstances() →exists(o <object-filter>)) and <SrcClass>.allInstances() →exists(o <object-filter> and o.(ref)→size() < <upB>))</code>	Wodel: create reference symbol to one Symbol in one Transition OCL: <code>Symbol.allInstances()→exists(s true) and Transition.allInstances() →exists(t t.symbol→size() < 1)</code>
Modificación de referencias: Existe una referencia no vacía del tipo dado, y más de un objeto del tipo destino de la referencia.	<code><SrcClass>.allInstances() →exists(o o.(ref)→notEmpty()) and <TgtClass>.allInstances() →size() > 1</code>	Wodel: modify target symbol from one Transition to other Symbol OCL: <code>Transition.allInstances() →exists(t t.symbol→notEmpty()) and Symbol.allInstances()→size() > 1</code>
Eliminación de referencias: Existe una referencia de la que se puede eliminar un objeto sin violar su cardinalidad inferior.	<code><Class>.allInstances() →exists(o <object-filter> and o.(ref)→size() > <lowB>))</code>	Wodel: t = select one Transition remove t→symbol OCL: <code>Transition.allInstances() →exists(t t.symbol→size() > 0)</code>

Tabla 3.1 Plantillas para generar las restricciones OCL a partir de las primitivas de mutación. plantilla (primera columna), y ejemplos de su aplicación (tercera columna). Por ejemplo, la plantilla OCL para la primitiva de borrado de objetos requiere de la existencia de un objeto con el tipo y los valores de atributos y referencias especificados en la primitiva, y que no esté contenido en ninguna referencia que viole su cardinalidad mínima si se borra el objeto.

La tabla muestra como ejemplo la eliminación de un objeto Symbol: la restricción OCL comprueba que existe al menos un objeto Symbol, y que el objeto Automaton en el que está contenido contiene al menos otro objeto en su referencia alphabet además del objeto Symbol que se elimina. Por ejemplo, si el número de elementos en la referen-

cia `Automaton.alphabet` es mayor que 1, al eliminar el objeto `Symbol` no se viola la cardinalidad inferior de la referencia.

El resto de plantillas OCL se corresponden con las primitivas de la creación de objetos (que requiere de la existencia de una referencia de contención compatible con el objeto creado y con espacio suficiente para el objeto), clonado de objetos (que además requiere de la existencia de un objeto candidato al que clonar), retipado de objetos (que requiere condiciones equivalentes a aquellas para eliminar y crear objetos para cada referencia, de contención o no, que no es compatible en origen o destino con el nuevo tipo), modificación de referencias (que requieren de la existencia de un objeto de la clase destino), creación de referencias (que además requieren una referencia con espacio suficiente para añadir el objeto de la clase destino), y borrado de referencias (que requieren que la referencia cumpla con su cardinalidad mínima tras eliminar uno de sus objetos).

Para facilitar la legibilidad, la Tabla 3.1 muestra la plantilla asociada a una única aplicación de cada primitiva de mutación. Sin embargo, un programa puede aplicar la misma primitiva con los mismos parámetros más de una vez. Esto puede suceder porque la primitiva está repetida, o porque se define un intervalo de aplicación superior a uno. Por tanto, en el caso general, se cuenta cuántas veces se aplica la misma instrucción, y se genera una restricción ligeramente más compleja, en la que cada una de estas apariciones de la instrucción se presenta como una variable. Por ejemplo, si la mutación `create CommonState` tiene que ejecutarse dos veces, se genera la siguiente invariante (ver Tabla 3.1):

```
Automaton.allInstances()→exists(a1,a2 |
  (a1 <> a2 and a1.states→size() < 10 and a2.states→size() < 10)
  or a1.states→size() < 9)
```

Como se ha explicado, el proceso de síntesis del modelo comienza con el meta-modelo del dominio y sus invariantes. A este meta-modelo de dominio se le añade una clase obligatoria auxiliar denominada `Dummy`. A continuación, el proceso genera las restricciones OCL para cada operador de mutación incluido en el programa `WODEL`, siguiendo las plantillas de la Tabla 3.1. Estas restricciones se añaden como invariantes de la clase `Dummy`. Finalmente, se lanza el buscador de modelos utilizando como entrada este meta-modelo enriquecido.

Como ejemplo, el Listado 3.8 muestra las restricciones OCL generadas para el programa `WODEL` del Listado 3.7. La restricción `mut1`, que se ha generado a partir de la primitiva `remove one CommonState` (línea 9 del Listado 3.7), requiere que en el modelo haya un objeto de tipo `CommonState`, y que además, la referencia `states` del objeto `Automaton` que la contiene guarde al menos otro objeto, de modo que al eliminar este objeto `CommonState` no se viole su cardinalidad inferior. La restricción `mut2` asegura que existe al menos un objeto de tipo `Transition` cuya referencia `src` es de tipo `FinalState` (línea 10 del Listado 3.7).

La restricción mut3 asegura que no se supere la cardinalidad máxima de la referencia contenedora states del objeto Automaton al crear el nuevo objeto FinalState (línea 12 del Listado 3.7). Por último, la invariante mut4 se genera al procesar la instrucción create Transition with {... symbol = one Symbol} (línea 13 del Listado 3.7) y asegura que existe al menos un objeto de tipo symbol en el modelo. La Figura 3.12 muestra el modelo semilla obtenido por el buscador de modelos. Este modelo semilla satisface las restricciones del Listado 3.8, y las incluidas en el meta-modelo original.

```

1 generate 3 mutants in "out/" from "models/"
2 metamodel "http://fa.com"
3 description "Wodel program example"
4
5 with blocks {
6   CNFSb "Removes a non-final and non-initial state, changes a final state to non-final
7   and connects it with a new final state"
8   {
9     remove one CommonState
10    t1 = select one Transition where {src is typed FinalState}
11    s2 = retype t1→src as CommonState
12    s3 = create FinalState
13    create Transition with {src = s2, tar = s3, symbol = one Symbol}
14  }
15 }

```

Listado 3.7 Un programa WODEL sencillo.

```

1 context Dummy
2
3 inv mut1 : CommonState.allInstances()→exists(s |
4   Automaton.allInstances()→forAll(a | a.states→includes(s) implies a.states→size() > 1))
5 inv mut2 : Transition.allInstances()→exists(t | t.src→oclIsKindOf(FinalState))
6 inv mut3 : Automaton.allInstances()→exists(a | a.states→size() < 10)
7 inv mut4 : Symbol.allInstances()→exists(s | true)

```

Listado 3.8 Restricciones derivadas del Listado 3.1.

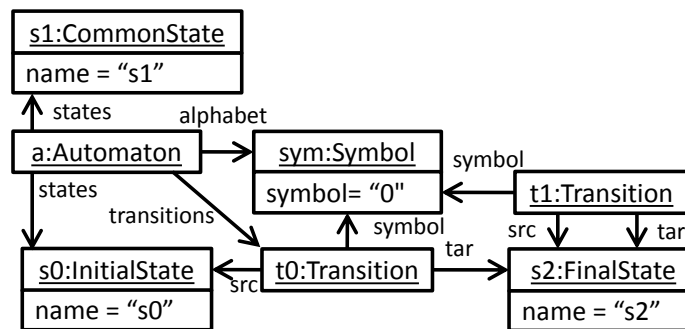


Figura 3.12 Modelo semilla generado.

Hay que indicar que los modelos semilla obtenidos con este enfoque permiten la aplicación de todas las instrucciones incluidas en el programa WODEL. No obstante, no se

garantiza que, después de ejecutar el programa, el mutante resultante satisfaga las restricciones incluidas en el meta-modelo. Esto requeriría de técnicas para anticipar las restricciones del meta-modelo a las operaciones del programa `WODEL`, usando técnicas similares a las presentadas en [33], cuestión que se deja como trabajo futuro.

3.3.2. Métricas de mutación

Con el objetivo de evaluar la cobertura del conjunto de operadores de mutación que se ha diseñado, `WODEL` permite el cálculo de métricas estáticas y dinámicas de los programas. Estas métricas son útiles para detectar partes del meta-modelo para las que no se ha diseñado ningún operador de mutación, o tipos de mutaciones que un programa `WODEL` no incluye. Se detallan las características de estas métricas en las siguientes subsecciones.

3.3.2.1. Métricas estáticas

Las métricas estáticas informan de los elementos del meta-modelo que se pueden ver afectados por un programa `WODEL`, y qué tipos de acciones (creación, modificación, y borrado) pueden realizarse sobre ellos. Estas métricas resultan de utilidad para analizar estáticamente si un conjunto de operadores de mutación es completo con respecto al meta-modelo, o al menos cubre la mutación de todos los elementos de interés del meta-modelo para un dominio de aplicación dado. Esta información resulta muy valiosa cuando el meta-modelo es grande o se consideran muchos operadores de mutación.

La información de métricas estáticas puede agregarse de acuerdo con dos criterios. Por un lado, las *métricas estáticas del meta-modelo* agregan la información para cada elemento del meta-modelo (clase, referencia o atributo). Por otro lado, las *métricas estáticas por operador* agregan la información por cada operador definido en el programa. De esta forma, las *métricas estáticas del meta-modelo* proporcionan información del número de veces que se crea, modifica o elimina cada elemento por alguna instrucción del código, y también calcula los porcentajes con respecto al número total de elementos. Las métricas estáticas por operador ayudan a entender los efectos colaterales de los operadores de mutación definidos, y cómo cada operador contribuye a la mutación general de los modelos.

De acuerdo con la información recopilada, se distingue entre acciones *explícitas* e *implícitas*. Las primeras son acciones definidas explícitamente en un programa `WODEL`, codificadas como primitivas de mutación de `WODEL` que actúan sobre las instancias de algún elemento del meta-modelo. Por ejemplo, la instrucción de mutación `remove one State from a.states` elimina de forma explícita un objeto de la referencia `Automaton.states`, y así, la métrica estática contaría una modificación explícita sobre esta referencia. Si hay otras

modificaciones explícitas de la referencia en el mismo programa, este contador se incrementaría en consecuencia. En cambio, las acciones implícitas son efectos colaterales sobre los modelos debidos a acciones explícitas. Por ejemplo, el operador de mutación `remove one Transition` elimina un objeto de tipo `Transition` de forma explícita, pero además, elimina de forma implícita sus referencias `Transition.src` y `Transition.tar`, y modifica su referencia contenedora `Automaton.transitions` al quitar la transición eliminada. De esta forma, esta información revela acciones ocultas de los operadores de mutación. El Listado 3.9 presenta un programa `WODEL` con el operador de mutación `RST` que incluye estas dos instrucciones, y la Tabla 3.2 muestra las métricas estáticas por meta-modelo correspondientes a este programa. Por ejemplo, la instrucción de la línea 9 del Listado 3.9 “`remove one State from a.states`” supone los borrados explícitos de la clase `State` y su atributo obligatorio `name`, y la modificación explícita de la referencia contenedora `a.states`. Estas acciones se muestran en la Tabla 3.2 en la fila correspondiente a la clase `State` en la columna de borrado explícito como “1c 1f”, es decir, una eliminación de clase (1c) y una eliminación de propiedad (1f). Este valor “1c 1f” se muestra también en la fila que corresponde a la clase `Automaton` en la columna de modificación explícita.

```

1 generate 2 mutants in "out/" from "models/"
2 metamodel "http://fa.com"
3 description "Wodel program example"
4
5 with blocks {
6   RST "Elimina un estado y una transicion aleatorios"
7   {
8     a = select one Automaton
9     remove one State from a.states
10    remove one Transition
11  }
12 }
```

Listado 3.9 Ejemplo de programa `WODEL`.

La Tabla 3.3 enumera las acciones implícitas que pueden suceder como resultado de una acción explícita. En particular, sólo la creación, borrado y clonación de objetos pueden conllevar acciones implícitas. De cara a computar estas acciones implícitas, las relaciones de herencia en el meta-modelo se tienen también en cuenta. Por ejemplo, cuando un operador especifica la creación de un objeto de tipo `C`, se incrementa el contador de creación implícita para todas las superclases de `C` porque los objetos de tipo `C` son compatibles con los ancestros de `C`. Igualmente, cuando un operador borra objetos de tipo `C`, se incrementa el contador de borrado implícito para todas las subclases de `C`, dado que el operador también puede borrar objetos de las subclases de `C`. Finalmente, como se ha explicado previamente, `WODEL` asigna automáticamente valores a las características obligatorias de los objetos creados para los que no se especifica un valor explícito. Por este motivo, crear un objeto incrementa el contador

Clases Ats./Refs.	Explícitas			Implícitas		
	C	M	D	C	M	D
Cobertura de clase	0 %	25 %	25 %	0 %	25 %	0 %
▼ Automaton		1c 1f			1c 1f	
(r) alphabet						
(r) states		1				
(r) transitions					1	
▼ State			1c 1f			
(a) name			1			
▼ Transition			1c 0f			0c 2f
(r) src						1
(r) tar						1
▷ Symbol						

Tabla 3.2 Métricas estáticas del meta-modelo para el operador de mutación definido en el Listado 3.9. C=Creación, M=Modificación, D=Borrado, (a)=atributo, (r)=referencia.

Acción explícita	Acciones implícitas derivadas
Creación de objeto de la clase C	<ul style="list-style-type: none"> ■ creación de la superclase de C ■ creación de atributos obligatorios definidos por C o un supertipo ■ creación de referencias obligatorias definidas por C o un supertipo ■ modificación de contenedores de C o un supertipo
Modificación de objeto de la clase C	<i>ninguna</i>
Borrado de objeto de la clase C	<ul style="list-style-type: none"> ■ borrado de subclases de C ■ borrado de atributos definidos por C o un supertipo ■ borrado de referencias definidas por C o un supertipo ■ modificación de referencias que apuntan a C o a un supertipo
Clonado de objeto de la clase C	<p><i>Clonado superficial:</i></p> <ul style="list-style-type: none"> ■ creación de superclases de C ■ creación de atributos obligatorios definidos por C o un supertipo ■ creación de referencias obligatorias definidas por C o un supertipo ■ modificación de contenedores de C o de un supertipo <p><i>Clonado completo:</i></p> <ul style="list-style-type: none"> ■ clonado superficial de C ■ para cada clase C' alcanzable desde C: clonado superficial de C'
Retipado de objeto de la clase C a un objeto de la clase T	<ul style="list-style-type: none"> ■ acciones implícitas para la creación de un objeto de clase T ■ acciones implícitas para la eliminación de un objeto de clase C
Creación de característica	<i>ninguna</i>
Modificación de característica	<i>ninguna</i>
Borrado de característica	<i>ninguna</i>

Tabla 3.3 Reglas para calcular las acciones implícitas a las que da lugar cada acción explícita.

de creación implícita de sus características obligatorias propias y heredadas, si no se ha indicado un valor explícito. El retipado de objetos se considera como la eliminación de un objeto seguida de la creación de un objeto, por lo tanto implica las correspondientes acciones implícitas. Finalmente, si un programa `WODEL` especifica una acción de forma explícita, esta acción se cuenta como explícita pero no como implícita.

3.3.2.2. Métricas dinámicas

Las métricas estáticas proporcionan una sobre-aproximación de los posibles efectos de un programa de mutación. Es una sobre-aproximación porque algunos efectos dependerán del modelo semilla concreto sobre el que se aplique el programa. Por ejemplo, cuando un operador de mutación borra un objeto, la métrica estática informa que todos los tipos de referencia que puede incluir el objeto son modificados de forma implícita. No obstante, en un modelo en concreto, un objeto puede estar contenido sólo en algunas de estas referencias.

Las métricas dinámicas permiten analizar los efectos reales que un programa de mutación tiene sobre modelos semilla específicos. Se distinguen dos tipos de métricas dinámicas. Por un lado, las *métricas dinámicas netas* resumen los efectos netos de un programa de mutación comparando el modelo semilla y el mutante resultante. Por otro lado, las *métricas dinámicas de depuración* son más detalladas dado que registran los operadores específicos aplicados por el programa. Esto resulta útil para detectar situaciones donde un operador de mutación cancela los efectos de operadores de mutación previos (p. ej., un operador crea un objeto, y un operador posterior lo borra). En estos casos, las métricas de depuración reflejarán los operadores de mutación cancelados, mientras que las métricas netas no proporcionan ese nivel de detalle. Esta información puede utilizarse para localizar errores en el programa identificando partes de los modelos semilla que no se han mutado como se esperaba.

3.4. Herramienta proporcionada

`WODEL` está disponible como plugin de Eclipse en <http://gomezabajo.github.io/Wodel/>. Incluye un editor desarrollado con Xtext [18] para la definición de programas `WODEL`. El editor proporciona resaltado de sintaxis, generación automática de código Java a partir de los programas `WODEL`, y comprobación de tipos en los programas en base al meta-modelo de dominio especificado, para asegurar que sólo se utilizan tipos y características válidos del meta-modelo. La tecnología subyacente es Eclipse Modeling Framework (EMF) [118], el estándar de-facto para modelado vigente en Eclipse.

La Figura 3.13 muestra la arquitectura modular y basada en componentes del entorno de mutación. El flujo de trabajo es como sigue. Primero, dado que `WODEL` es independiente del

dominio, el usuario ha de describir los conceptos del dominio por medio de un meta-modelo Ecore (etiqueta 1 en la figura). Por ejemplo, si el usuario quiere mutar autómatas finitos debe proporcionar un meta-modelo que incluya los conceptos de estado, transición y símbolo del alfabeto, así como las relaciones existentes entre estos elementos.

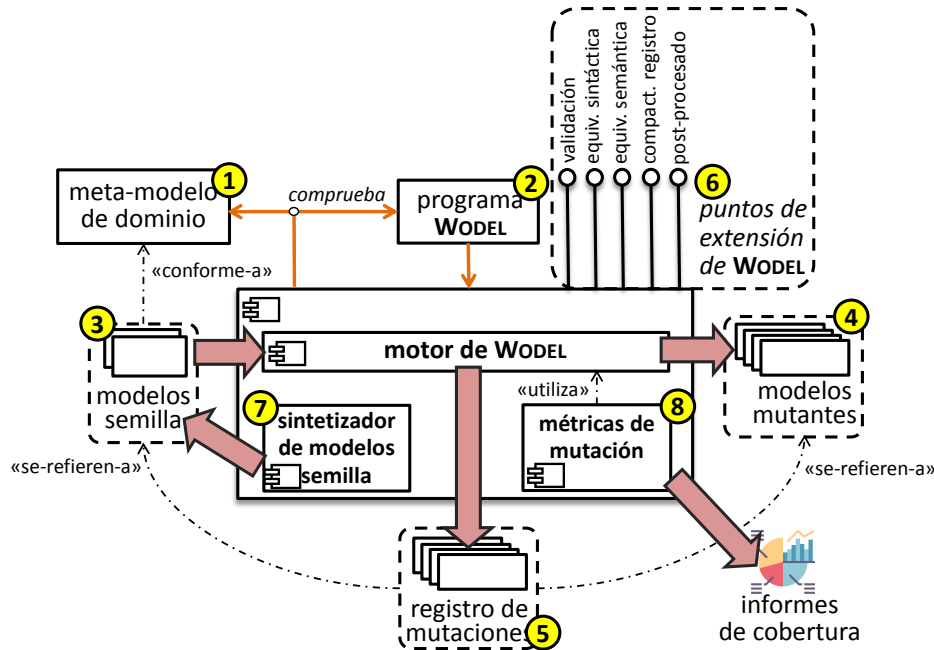


Figura 3.13 Arquitectura del entorno de desarrollo de WODEL.

A continuación, el usuario define los operadores de mutación deseados y los detalles de su ejecución, como el número de veces que debe aplicarse cada operador de mutación, o el orden de ejecución de los operadores. A esta especificación la denominamos programa WODEL (etiqueta 2).

La ejecución de un programa WODEL produce modelos mutantes a partir de los modelos semilla (etiqueta 4 en la Figura 3.13), así como un registro de los operadores de mutación aplicados y los objetos a los que afectan (etiqueta 5 en la Figura 3.13, ver Sección 3.2.2).

Una vez creado, un programa WODEL puede aplicarse a los modelos semilla que proporcione el usuario, los cuales deben ser conformes al meta-modelo del dominio dado (etiqueta 3 en la Figura 3.13). Adicionalmente, el usuario puede utilizar el sintetizador de modelos semilla que incorpora WODEL para generar automáticamente modelos de prueba a partir del programa WODEL dado, tal como se explica en la Sección 3.3.1 (etiqueta 7). También puede recopilar métricas de mutación del programa con objeto de detectar operadores no aplicados o elementos del meta-modelo no cubiertos (etiqueta 8).

WODEL proporciona cinco puntos de extensión de su funcionalidad, basándose en el mecanismo de puntos de extensión que proporciona Eclipse (etiqueta 6 en la Figura 3.13).

El primer punto de extensión permite personalizar la validación de los mutantes (ver Sección 3.2.1). Los puntos de extensión segundo y tercero habilitan la configuración del criterio de equivalencia de los mutantes, distinguiendo la equivalencia sintáctica y la semántica (ver Sección 3.2.3). El cuarto punto de extensión permite la personalización y ampliación de la compactación del registro de mutaciones aplicadas (ver Sección 3.2.2). El quinto y último punto de extensión permite registrar post-procesadores que generen artefactos de dominio específico dirigidos a aplicaciones particulares. Este punto de extensión proporciona una interfaz Java con los siguientes métodos que el usuario que quiere crear una aplicación de post-procesado de `WODEL` ha de completar.

- `getName`: Se ha de incluir aquí el nombre que identifica la aplicación, de forma que se puede distinguir de entre todas las aplicaciones de post-procesado creadas.
- `doGenerate`: Este método se utiliza para crear el proyecto de la aplicación de post-procesado. Tiene como parámetros el nombre del programa `WODEL`, el meta-modelo de dominio, el nombre del proyecto, y las carpetas de entrada y de salida.
- `doRun`: Este método se utiliza para ejecutar propiamente la aplicación de post-procesado sobre los mutantes generados por `WODEL`. En el caso de `WODEL-EDU`, se generarán los ejercicios que se corrigen de forma automática; y en el caso de `WODEL-TEST`, se realizará el proceso de pruebas de mutación.

El primer paso que debe realizar un usuario para utilizar `WODEL` es instalarse el plugin de Eclipse desde el update-site <http://gomezabajo.github.io/Wodel/update-site/>. A continuación, el usuario ha de crear un proyecto nuevo `WODEL`. El asistente guía al usuario en la configuración del proyecto, en la que se debe indicar un nombre de proyecto, el nombre del fichero que contendrá el programa `WODEL`, la carpeta donde se ubicarán el meta-modelo de dominio y los modelos semilla, y la carpeta de salida en la que se guardarán los modelos mutantes. Dentro de dicha carpeta de salida, los mutantes se guardarán en la carpeta indicada siguiendo una jerarquía basada en los bloques del programa `WODEL`.

La Figura 3.14 presenta la página de preferencias de `WODEL`. En esta página se puede configurar la generación del registro de operadores de mutación aplicados, que se guarda en una carpeta `registry` dentro de la carpeta en la que se ubican los mutantes a los que corresponde.

También se puede deshabilitar la serialización de los modelos, de forma que se pueden utilizar desde memoria para la aplicación de post-procesado. Se puede habilitar una extensión de post-procesado para generar los mutantes en un formato distinto a EMF XMI (por ejemplo, json). La página de preferencias permite configurar el número máximo de intentos

para conseguir un mutante válido, y el número máximo de mutantes que se generarán por defecto (que se utilizará si no se ha incluido este dato en el programa WODEL). Por último, en esta página se puede configurar también la extensión para validar los mutantes, descartar los mutantes equivalentes sintácticamente, y aquellos equivalentes semánticamente.

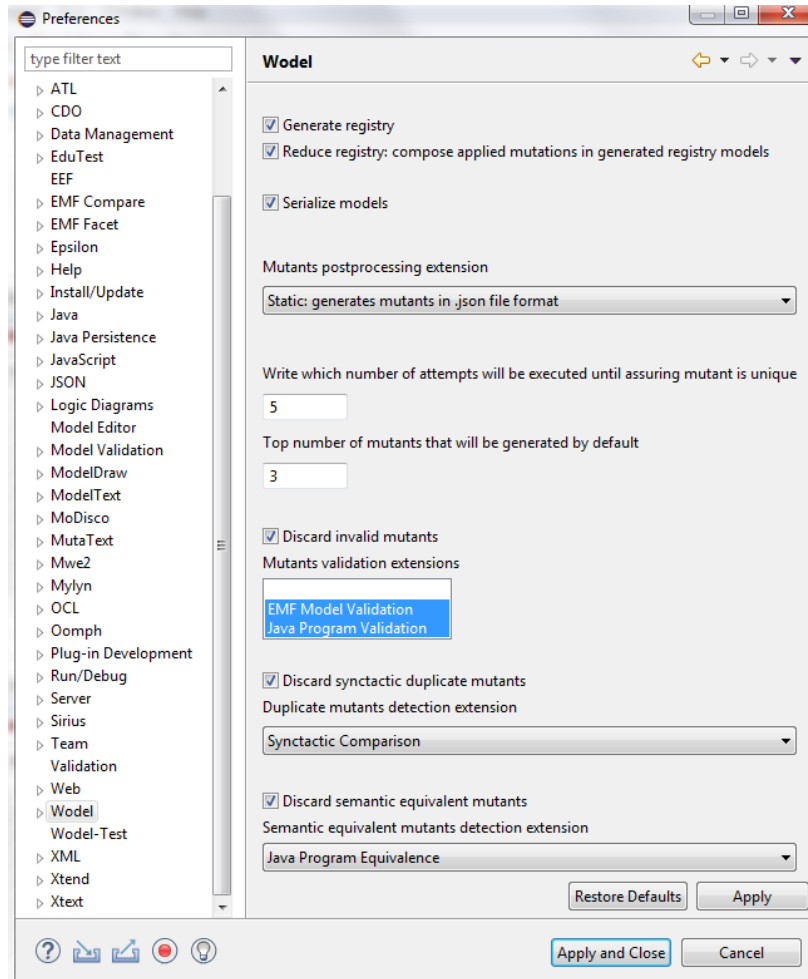


Figura 3.14 Página de preferencias de WODEL.

La síntesis de modelos semilla para un programa dado se puede configurar por medio del asistente que se muestra en la Figura 3.15. Este asistente permite establecer el número máximo de modelos semilla a generar, los operadores de mutación utilizados en el proceso de generación de modelos (o bien, todos los operadores incluidos en el programa, o bien, un subconjunto de los mismos), requisitos adicionales del modelo expresados mediante OCL, y opcionalmente, un modelo EMF que se utilizará como modelo de partida en la búsqueda de modelos. Además, una página de preferencias permite configurar el mínimo y el máximo

número de objetos y referencias que tendrán los modelos semilla generados. La búsqueda de modelos semilla se realiza utilizando el buscador de modelos USE Validator [74].

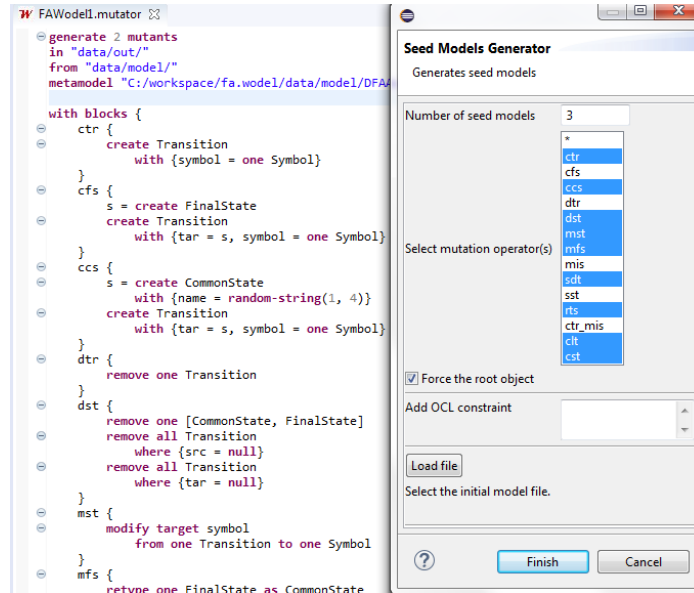


Figura 3.15 Asistente de WODEL para la generación de modelos semilla.

La Figura 3.16 es una captura de pantalla de WODEL que muestra su editor (etiqueta 1), el código Java generado a partir de un programa WODEL (etiqueta 2), un meta-modelo de dominio y algunos modelos semilla (etiqueta 3), varios mutantes generados a partir de los modelos semilla (etiqueta 4), y las métricas estáticas y dinámicas del programa (etiqueta 5). Los diferentes artefactos en la captura de pantalla corresponden al ejemplo en el dominio de los autómatas finitos.

WODEL permite visualizar las métricas de mutación de un programa WODEL sobre el diagrama del meta-modelo (Figura 3.17) o utilizando vistas dedicadas de Eclipse con tablas desplegadas (Figura 3.16, etiqueta 5). Las vistas muestran la información organizada por el elemento del meta-modelo, por el operador de mutación, y sus celdas utilizan colores diferentes para distinguir de forma sencilla entre las mutaciones de creación, modificación y borrado.

La vista de las métricas estáticas de la Figura 3.16 (esquina superior izquierda de la etiqueta 5) presenta el recuento del número de creaciones explícitas e implícitas (C, IC), de modificaciones (M, IM) y de eliminaciones (D, ID), presentadas por cada elemento del meta-modelo, o efectuadas por cada operador de mutación. Las celdas correspondientes a las clases agregan las acciones realizadas en la clase, y en sus atributos y referencias. Por ejemplo, la celda para la creación explícita de la clase Transition contiene 6c 11f porque el programa contiene seis creaciones explícitas de objetos de tipo Transition y

once creaciones explícitas de sus atributos y referencias. La primera fila de la vista muestra el promedio de mutantes que crean, modifican o borran objetos de alguna clase, respectivamente. Por ejemplo, el porcentaje de creación explícita es de un 57 % porque el programa crea explícitamente 4 de las 7 clases concretas del meta-modelo de dominio.

Las métricas dinámicas de la derecha de la Figura 3.16 presentan columnas que indican el número de elementos que realmente se crean (C), modifican (M) o eliminan (D) por la ejecución del programa `WODEL` sobre el modelo semilla (`binaryfa.model`), y los efectos que cada operador de mutación tiene sobre el modelo. La primera fila de la tabla muestra el promedio de mutantes en los que se ha creado, modificado o eliminado algún objeto.

The screenshot displays the WODEL environment within the Eclipse IDE. The interface includes a Project Explorer on the left showing the project structure, a main editor window displaying the `FAWModel1.mutator` file, and a 'Model Static Metrics Class Wizard' dialog box. The dialog box is titled 'Model Mutation Creation Tool. Class name: Symbol' and shows a list of mutation operators (Mutators) and a selected strategy (RandomTypeSelection). The main editor shows the following code:

```
generate 2 mutants
in "data/out/"
from "data/model/"
metamodel "C:/workspace/fa.w
with blocks {
  ctr {
    create Transition wi
  }
  cfs {

```

The right side of the interface displays four tables showing static and dynamic footprints. The 'Meta-model Static Footprint' table shows metrics for various classes and attributes. The 'Net Dynamic Footprint' table shows metrics for models, blocks, and mutants. The 'Operator Static Footprint' table shows metrics for blocks, commands, and operators. The 'Debug Dynamic Footprint' table shows metrics for models, blocks, and mutants.

Figura 3.16 El entorno WODEL en acción.

Como se ha mencionado previamente, las métricas estáticas del meta-modelo se pueden visualizar también sobre el meta-modelo (específicamente, sobre una visualización basada en Sirius [40] del meta-modelo Ecore de dominio). La Figura 3.17 muestra el meta-modelo del ejemplo con anotaciones de la información de las métricas. Utilizamos una metáfora de semáforo en la que las clases y los atributos incluyen iconos con colores diferentes para indicar la creación (verde), la modificación (ámbar), y la eliminación (rojo) explícitas de sus instancias. Estos colores son los mismos que se utilizan en las vistas de las métricas (ver Figura 3.16). En el ejemplo, sólo las clases *Transition*, *InitialState*, *CommonState* y *FinalState* se mutan de forma explícita, y por tanto, sólo se anotan estas clases con iconos. De forma similar, nuestra visualización muestra las referencias mutadas de forma explícita con un color diferente. En este ejemplo, se modifican de forma explícita las referencias *symbol*, *src* y *tar* de los objetos de tipo *Transition*.

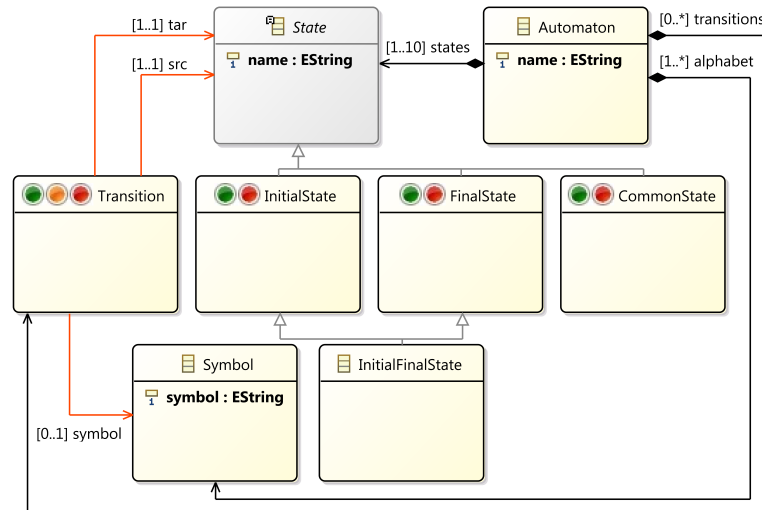


Figura 3.17 Meta-modelo con la métrica estática.

En el caso en que las métricas revelen que un programa de mutación no proporciona una cobertura completa del meta-modelo de dominio, la herramienta *WODEL* permite la creación semi-automática de operadores de mutación que mejoren esta cobertura. Con este objetivo, la vista de las métricas estáticas del meta-modelo integra un asistente que puede activarse haciendo doble-clic en una clase del meta-modelo o en el nombre de una característica. Este asistente permite seleccionar una de las primitivas de mutación para resolver la falta de cobertura, y configurar a la vez sus detalles de ejecución. El asistente extiende de forma automática el programa *WODEL* original con el nuevo operador de mutación. La etiqueta 6 de

la Figura 3.16 muestra la utilización del asistente para crear un operador de mutación sobre la clase del meta-modelo `Symbol` que aún no ha sido cubierta.

3.5. Evaluación de la concisión de Wodel

Uno de los aspectos que determina la simplicidad de un lenguaje es su concisión. Dado que uno de los objetivos de `WODEL` es simplificar la definición de operadores de mutación, en esta sección presentamos un análisis de la concisión de `WODEL` comparándolo con el código Java que sería necesario para realizar una implementación equivalente de un programa de mutación.

Programar los operadores de mutación en Java requiere conocimientos de la API reflexiva de EMF, dado que no se puede asumir que existe una implementación en Java de los tipos del meta-modelo dado. También se requiere tener en cuenta detalles accidentales que `WODEL` gestiona por defecto, como colocar los objetos en contenedores, inicializar las referencias obligatorias, comprobar los tipos de los operadores de mutación con respecto al meta-modelo, serializar los modelos, comprobar la validez de los mutantes resultantes, comparar la equivalencia de los mutantes, o producir un registro con los operadores de mutación aplicados.

Para ilustrar la complejidad del código Java equivalente, el Listado 3.11 muestra parte del código que implementa el operador de mutación del Listado 3.10. Este fragmento de código Java crea una transición (líneas 3-4), obtiene un objeto autómatas a partir del modelo semilla (líneas 8-10), añade la transición al autómatas (líneas 12-13), selecciona un estado aleatorio (líneas 15-17), y establece el estado como origen de la transición (líneas 18-19). El listado omite la asignación de un estado destino de la transición y un nombre, que también son características obligatorias. Además, tampoco incluye el código de tareas como la carga del modelo y del meta-modelo y la comprobación de la validez del resultado. En total, el operador de mutación ocupa 103 líneas de código, excluyendo líneas vacías y comentarios. Esta misma funcionalidad se obtiene utilizando 1 línea de código en `WODEL`.

```
1 generate 2 mutants in "out/" from "model/"
2 metamodel "http://fa.com"
3
4 with commands {
5   t = create Transition
6 }
```

Listado 3.10 Operador de mutación que crea una transición.

```

1 ...
2 // create transition
3 EClass transitionClass = (EClass)epackage.getEClassifier("Transition");
4 EObject transition = EcoreUtil.create(transitionClass);
5
6 // search object automaton in model
7 EObject automaton = null;
8 for (TreeIterator<EObject> it = seed.getAllContents(); it.hasNext(); ) {
9     automaton = it.next();
10    if (automaton.eClass().getName().equals("Automaton")) {
11        // add transition to automaton
12        EStructuralFeature feature = automaton.eClass().getEStructuralFeature("transitions");
13        ((List<EObject>)automaton.eGet(feature)).add(transition);
14        // set random state as source of the transition
15        feature = automaton.eClass().getEStructuralFeature("states");
16        List<EObject> states = (List<EObject>)automaton.eGet(feature);
17        EObject randomState = states.get(rand.nextInt(states.size()));
18        feature = transitionClass.getEStructuralFeature("src");
19        transition.eSet(feature, randomState);
20 ...

```

Listado 3.11 Código Java para el operador de mutación del Listado 3.10.

La Tabla 3.4 presenta una revisión del número de líneas de código Java necesarias para implementar cada una de las primitivas de mutación proporcionadas por el DSL *WODEL*. En esta tabla se presenta el número de líneas de código Java de la función de utilidad que implementa la mutación en sí, que incluye además el código Java necesario para guardar la información utilizada por el registro de mutaciones (columna 3), el número de líneas para la inicialización de atributos y referencias (columna 4), el número de líneas necesarias para añadir el objeto a un contenedor (columna 5), y el número de líneas para copiar los atributos y referencias compatibles, como en la primitiva de retipado (columna 6). Por ejemplo, para la primitiva de mutación de retipado son necesarias 36 LOC Java para la mutación en sí, 26 LOC para la inicialización de atributos y referencias, 39 LOC para añadir el nuevo objeto a un contenedor, que son necesarias en el caso de que la referencia contenedora del objeto que se ha retipado no sea compatible con el nuevo objeto, y 13 LOC para copiar los atributos y referencias compatibles.

El número de LOC que se necesita para la creación de referencias (40) corresponde a la selección de un destino aleatorio de la referencia, la comprobación de que el tipo destino es compatible con el tipo del objeto, y las operaciones necesarias para crear la referencia en caso de que sea de tipo valor múltiple o simple. En el caso de la modificación de referencias, el código necesario consta de 67 LOC, que corresponden a la selección de un origen y un destino aleatorios de la referencia que se quiere modificar, la selección del nuevo origen o destino indicado en la instrucción de mutación, la comprobación de que el tipo del nuevo origen/destino es compatible con el tipo origen/destino de la referencia y que además tiene

espacio suficiente para el nuevo objeto, y la modificación en sí de la referencia en caso de que sea de tipo valor múltiple o simple.

La función de utilidad que implementa la primitiva de clonado superficial de objetos consta de 75 LOC (10+26+39), que corresponden a la creación del objeto, la copia del valor de los atributos y referencias del objeto original al clon, y la búsqueda de un contenedor compatible. En el caso de la primitiva de clonado completo de objetos, el código Java de la función de utilidad que la implementa consta de 109 LOC (44+26+39), dado que hay que crear un objeto, copiar el valor de los atributos y referencias no contenedoras del objeto original al clon, buscar un contenedor, y crear clones de los objetos contenidos directa e indirectamente en el objeto original. Éste último paso puede verse como un proceso recursivo.

En el caso de la eliminación de objetos, el código Java de la función de utilidad que la implementa consta de 29 LOC, mediante el que se efectúa el borrado del objeto especificado en la instrucción, y se elimina dicho objeto de las referencias en las que estaba incluido, tanto si son de tipo valor múltiple, como si son de valor simple. Por último, el código Java de la función de utilidad que implementa la primitiva de eliminación de referencias consta de 26 LOC, que corresponden a la selección de la referencia, y la eliminación de la referencia en sí, que puede ser de tipo valor múltiple o simple.

Mutación	Primitiva WODEL	LOC en Java	Inicialización de atributos y referencias	Añadir a contenedor	Copia de atributos y referencias
Selección de objetos	select one (Class)	7	-	-	-
Modificación de objetos	modify one (Class)	7 (Selección de objs.)	26	-	-
Creación de objetos	create (Class)	10	26	39	-
Eliminación de objetos	remove one (Class)	29	-	-	-
Clonado superficial de objetos	deep clone one (Class)	10	26	39	-
Clonado completo de objetos	deep clone one (Class)	44	26	39	-
Retipado de objetos	retype one (SrcClass) as (TarClass)	36	26	39	13
Creación de referencias	create reference (ref) to one (TarClass) in one (SrcClass)	40	-	-	-
Modificación de referencias	modify target (ref) from one (SrcClass) to one (TarClass)	67	-	-	-
Eliminación de referencias	c = select one (Class) remove c → (ref)	26	-	-	-

Tabla 3.4 Líneas de código Java requeridas para implementar cada primitiva de mutación WODEL.

En este capítulo hemos presentado el DSL WODEL, su sintaxis y las funcionalidades adicionales de la herramienta que se proporciona para desarrollar con este lenguaje. A continuación, los Capítulos 4 y 5 muestran dos aplicaciones construidas sobre WODEL que utilizan mutantes generados a partir de modelos semilla para la generación automática de

ejercicios de auto-evaluación (Capítulo 4) y la creación de entornos de pruebas de mutación para lenguajes de modelado y programación (Capítulo 5).

Capítulo 4

Wodel-Edu: Generación Automática de Ejercicios

En este capítulo se describe el post-procesador de WODEL para la generación automática de ejercicios al que hemos llamado WODEL-EDU. Al igual que WODEL, WODEL-EDU puede aplicarse a cualquier meta-modelo de dominio, lo que permite generar ejercicios en diversos ámbitos (p. ej. ejercicios de autómatas, circuitos digitales, diagramas de clases, etc.). Recibe como entrada el modelo solución a un ejercicio dado, y utiliza esta solución como modelo semilla al que aplicar una serie de mutaciones con objeto de obtener soluciones incorrectas. A continuación, genera una aplicación web con distintos tipos de ejercicios que utilizan el modelo solución o semilla y sus mutantes. Los ejercicios generados pueden calificarse de manera automática para la auto-evaluación del estudiante. En concreto, WODEL-EDU da soporte a la generación de tres tipos de ejercicios de complejidad incremental (ver los distintos esquemas de generación en la Figura 4.1):

- *Respuesta alternativa:* Estos ejercicios muestran un diagrama, y los estudiantes deben decidir si es correcto o no. Éste es el tipo más sencillo de ejercicio, en el que el diagrama es correcto si corresponde al modelo semilla, o incorrecto si corresponde a uno de los mutantes (ver Figura 4.1a).
- *Selección de diagrama múltiple:* Estos ejercicios muestran varios diagramas entre los que sólo uno es correcto, y los estudiantes deben identificarlo. Por tanto, este tipo de ejercicio sólo tiene que presentar el diagrama correspondiente al modelo semilla sin mutaciones entre las versiones mutadas (ver Figura 4.1b).
- *Selección de reparación múltiple:* Este es el tipo más complejo de ejercicios. En este caso, el ejercicio muestra un diagrama incorrecto generado al aplicar uno o varios

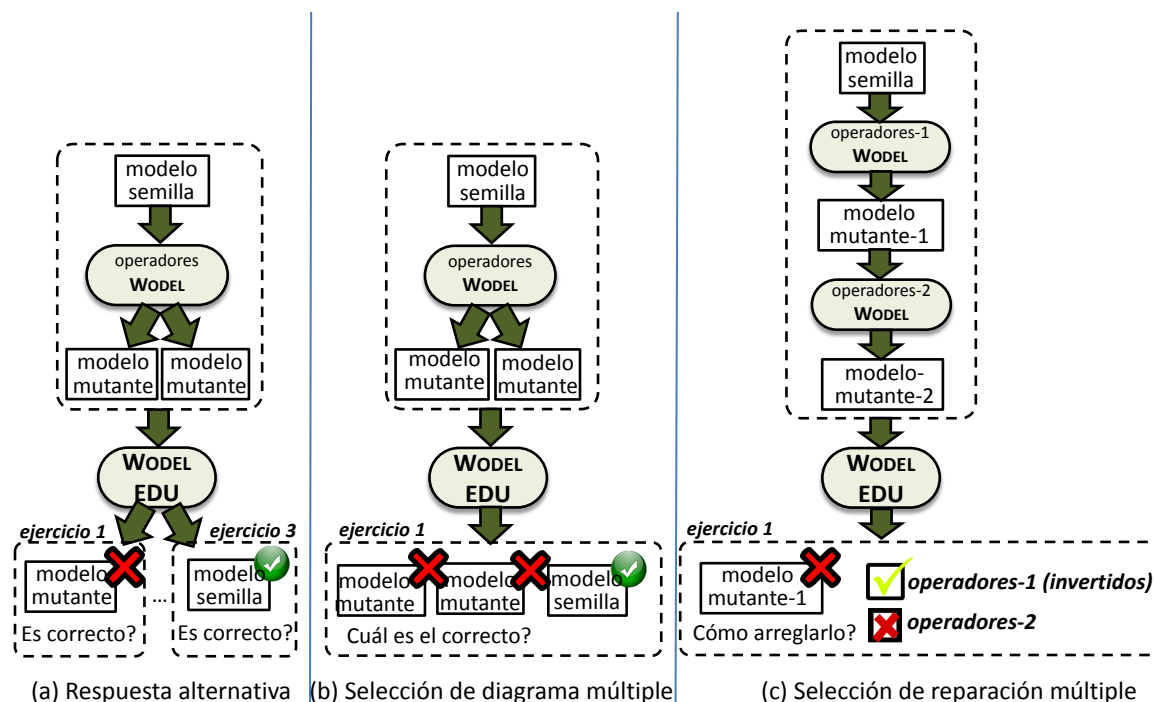


Figura 4.1 Esquemas de generación de los tres tipos de ejercicios.

operadores de mutación sobre el modelo semilla, así como la descripción de varias posibles reparaciones sobre el diagrama, y los estudiantes deben elegir el subconjunto de reparaciones que corregirían el diagrama. Las reparaciones correctas se sintetizan automáticamente invirtiendo los operadores de mutación aplicados para generar el diagrama incorrecto. Las opciones de reparación incorrectas se generan mutando el diagrama incorrecto con el objetivo de obtener acciones de modificación que tengan sentido para el ejercicio presentado (ver Figura 4.1c).

En el resto de este capítulo, la Sección 4.1 comienza presentando la arquitectura de la herramienta WODEL-EDU. A continuación, la Sección 4.2 describe la familia de DSLs utilizados por WODEL-EDU para la configuración y generación de los ejercicios. La Sección 4.3 ilustra los tres tipos de ejercicios generados por WODEL-EDU para el dominio de autómatas finitos. Por último, la Sección 4.4 presenta los resultados de un estudio con usuarios donde se evalúa la calidad de los ejercicios que genera WODEL-EDU.

4.1. Arquitectura

WODEL-EDU requiere configurar diferentes aspectos de la generación de ejercicios. En primer lugar, el meta-modelo del dominio de los ejercicios y el tipo de errores (o mutaciones)

que se introducirán en las soluciones. En segundo lugar, el tipo de ejercicio que se quiere generar y una descripción textual con el enunciado de cada ejercicio. En tercer lugar, cómo se representarán los modelos gráficamente, ya que esto dependerá del dominio de los ejercicios (p. ej., autómatas, diagramas de clase, etc.). Finalmente, para los ejercicios de selección de reparación múltiple, el texto de las diferentes opciones de reparación se genera a partir de los operadores de mutación aplicados, y por tanto, puede ser necesario afinar este texto.

La Figura 4.2 muestra la arquitectura del post-procesador **WODEL-EDU**. En primer lugar, **WODEL-EDU** requiere proporcionar el meta-modelo del dominio de los ejercicios y un programa **WODEL** con la especificación de los operadores de mutación. Además, **WODEL-EDU** proporciona cuatro DSLs para configurar el texto y el estilo de los ejercicios (**EDUTEST**), cómo los elementos del modelo deben representarse gráficamente (**MODELDraw**), cómo representar un elemento del modelo textualmente (**MODELText**), y cómo expresar los operadores de mutación en lenguaje natural (**MUTAText**). Estos DSLs facilitan la configuración de los ejercicios generados para dominios diferentes (p. ej., autómatas, diagramas de clase, circuitos electrónicos, etc.). Si bien **EDUTEST** y **MODELDraw** deben utilizarse siempre para configurar cualquiera de los tipos de ejercicios, los DSLs **MODELText** y **MUTAText** se utilizan únicamente para los ejercicios del tipo *selección de reparación múltiple* cuando es necesario sobrescribir el texto de las reparaciones que **WODEL-EDU** sintetiza por defecto.

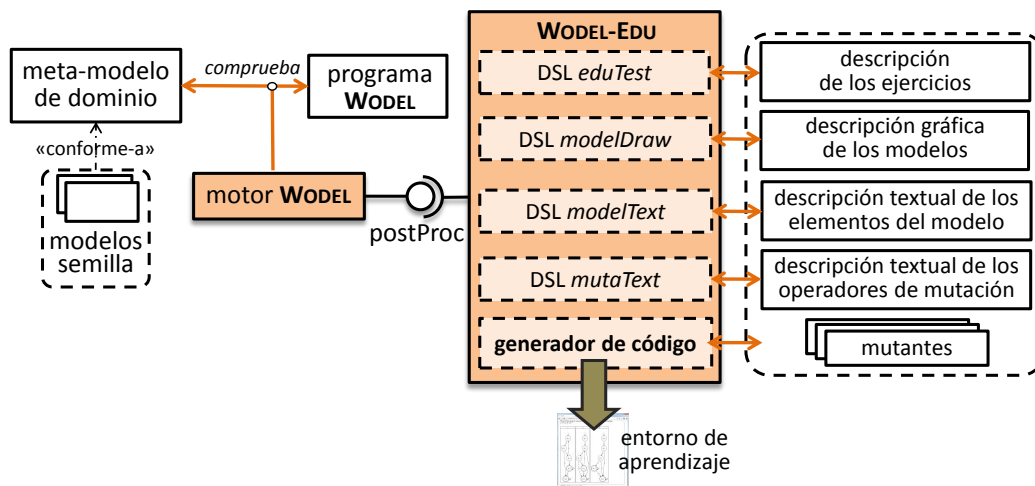


Figura 4.2 Arquitectura de **WODEL-EDU**.

El generador de código de **WODEL-EDU** recibe estas cuatro especificaciones, las aplica sobre los modelos semilla y los mutantes, genera los diagramas gráficos con la sintaxis concreta de los mismos, y el código html y Javascript de una aplicación web de ejercicios sobre el dominio dado que se corrigen automáticamente.

4.2. Lenguajes de configuración de ejercicios

En esta sección se describen los DSLs `EDUTEST`, `MODELDraw`, `MODELTEXT` y `MUTATEXT` que proporciona `WODEL-EDU` para configurar la generación de ejercicios. A modo de ejemplo, se utilizará el dominio de los autómatas finitos para ilustrar las características de cada uno de estos lenguajes.

4.2.1. Describiendo los ejercicios con `EDUTEST`

El DSL `EDUTEST` permite describir y configurar los ejercicios que se quieren generar. Como se ha indicado previamente, los ejercicios pueden ser de tres tipos: `AlternativeResponse`, `MultiChoiceDiagram` y `MultiChoiceEmendation`. La especificación de `EDUTEST` incluye el enunciado de cada ejercicio, y genera de forma automática el correspondiente código html y Javascript necesario.

`EDUTEST` permite configurar si se quiere permitir que los estudiantes naveguen libremente dentro de la aplicación de ejercicios, para p. ej., regresar a un ejercicio previo utilizando la directiva `navigation=free`, o bien, si se desea bloquear esta característica, con la directiva `navigation=locked`. Para cada una de las páginas de ejercicios, `EDUTEST` permite al estudiante realizar todos los intentos que necesite para resolver los ejercicios con la directiva `retry=yes`, o un único intento, con la directiva `retry=no`.

Otra característica configurable mediante `EDUTEST` es asignar distinto peso a cada uno de los ejercicios con la directiva `weighted=yes`. Este peso es proporcional a la cantidad de opciones que se presentan para resolver cada ejercicio. En otro caso, si se quiere que todas los ejercicios tengan el mismo peso, se debe incluir la directiva `weighted=no`.

Adicionalmente, `EDUTEST` permite configurar las siguientes características: el peso que tendrá cada fallo en la nota final utilizando la directiva `penalty=[porcentaje que se penaliza]`; el orden de los ejercicios, que puede ser fijo según el orden con el que se definen en la especificación `EDUTEST` (`order=fixed`), aleatorio (`order=random`), con un orden ascendente según el número de opciones para resolver cada ejercicio (`order=options-ascending`), o con un orden descendente (`order=options-descending`). Por último, se permite configurar que los ejercicios sean de respuesta única, con la directiva `mode=radiobutton`, o de forma alternativa, de respuesta múltiple, con la directiva `mode=checkbox`, en los que cada reparación requerida se muestra por separado en una casilla de verificación, que puede requerir que se seleccionen una o más para resolver el ejercicio correctamente.

El Listado 4.1 muestra una parte con la descripción de algunos ejercicios utilizando el DSL `EDUTEST`. La Línea 1 establece que los ejercicios generados pueden resolverse en cualquier orden. Las Líneas 3-8 definen dos ejercicios de *selección de reparación múltiple*

con las características siguientes: el reintento se permite en caso de fallo (`retry=yes`); todos los ejercicios tienen el mismo peso en el grado final (`weighted=no`); no hay penalización en caso de fallo (`penalty=0.0`); los ejercicios se muestran en orden descendente según el número de opciones de reparación (`order=options-descending`); todas las reparaciones requeridas para arreglar un diagrama se agrupan como opción única (`mode=radiobutton`). Las líneas 6-7 especifican el enunciado del ejercicio que se genera desde el modelo semilla dado. Debajo, las líneas 10-13 definen un ejercicio del tipo *selección de diagrama múltiple*. Por último, las líneas 15-19 establecen la configuración para un ejercicio del tipo *respuesta alternativa*.

```

1 navigation=free
2
3 MultiChoiceEmendation complex {
4   retry=yes, weighted=no, penalty=0.0,
5   order=options-descending, mode=radiobutton
6   description for 'exercise4.model' = 'Select the required changes so that the automaton accepts a+b+'
7   description for 'exercise6.model' = 'Select the required changes so that the automaton accepts a*b'
8 }
9
10 MultiChoiceDiagram simple1 {
11   retry=no
12   description for 'exercise1.model' = 'Select which of these automata accepts the language a*bab*'
13 }
14
15 AlternativeResponse simple2 {
16   retry=no
17   description for 'exercise8.model' = 'Does this automaton accept the language "ab(ba)*"?'
18   description for 'exercise9.model' = 'Does this automaton accept the language "(ab)*ba"?'
19 }

```

Listado 4.1 Definiendo los ejercicios con `EDUTEST`.

4.2.2. Describiendo la visualización de los modelos con `MODELDRAW`

La representación del modelo se configura utilizando el DSL `MODELDRAW`. Éste es un lenguaje sencillo similar a la notación *dot* proporcionada por *Graphviz*¹, que es la tecnología utilizada por `WODEL-EDU` para visualizar los modelos. `MODELDRAW` permite configurar, para cada clase del meta-modelo de dominio, si se mostrará como cierto tipo de nodo (círculo, círculo doble, elipse, rectángulo, etc.) o como una conexión. Los nodos y las conexiones pueden mostrar una etiqueta con el valor de algún atributo de la clase correspondiente. Por convención, si no se proporciona una etiqueta, se muestra el contenido del atributo `name` (si existe). También se da soporte a diferentes visualizaciones de la misma clase dependiendo del valor de algún atributo booleano.

¹<http://www.graphviz.org/>

El Listado 4.2 muestra un ejemplo de la utilización de `MODELDraw` para describir la apariencia gráfica de los autómatas finitos. La primera línea especifica el meta-modelo de dominio, lo que permite proporcionar asistencia de contenido y comprobación de tipos en el resto del programa. A continuación, el listado declara que los estados iniciales se mostrarán como círculos con una marca (línea 4), los estados no finales como círculos (línea 5), los estados finales como círculos dobles (línea 6), los estados iniciales y finales como círculos dobles con una marca (línea 7), y las transiciones como conexiones etiquetadas con el símbolo de la transición (línea 8).

```
1 metamodel "http://fa.com"
2
3 Automaton: diagram {
4   InitialState: markednode
5   CommonState: node, shape=circle
6   FinalState: node, shape=doublecircle
7   InitialFinalState: markednode, shape=doublecircle
8   Transition (src, tar): edge, label=symbol
9 }
```

Listado 4.2 Definiendo la representación gráfica de los modelos con `MODELDraw`.

4.2.3. Descripción textual de los elementos del modelo con `MODEL-TEXT`

Por defecto, cuando `WODEL-EDU` tiene que generar una descripción textual de un objeto, el objeto se describe por el valor de su atributo `name` si existe, o por el nombre de su clase en otro caso. El DSL `MODELTEXT` permite sobrecribir este texto por defecto para los tipos seleccionados del meta-modelo. De esta forma, para cada clase y relación, se puede especificar una plantilla de texto en la que las expresiones precedidas por el símbolo `%` se evalúan sobre el objeto y su valor se emite en el string resultante.

El Listado 4.3 muestra la utilización del DSL para el ejemplo de autómatas finitos. De acuerdo con la línea 3, cuando un objeto de tipo `State` se menciona en algún texto, se escribirá como “State” seguido por el nombre del estado (p. ej., “State q0” si el nombre del estado es “q0”). De forma similar, la línea 4 define que los objetos de tipo `Transition` se representarán por el texto “Transition” seguido por el símbolo de la transición. Este símbolo se obtiene al navegar a través de la referencia `symbol` del objeto `Transition`, y a continuación accediendo al atributo `symbol` (ver el meta-modelo de autómatas de la Figura 2.2). Finalmente, en las líneas 5-6, las referencias `src` y `tar` de los objetos de tipo `Transition` se configuran para presentarse con el texto “source” y “target”.

```
1 metamodel "http://fa.com"
2
3 > State: State %name
4 > Transition: Transition %symbol.symbol
5 > Transition.src: source
6 > Transition.tar: target
```

Listado 4.3 Definiendo la descripción textual de los elementos del modelo con `MODELTEXT`.

4.2.4. Descripción textual de los operadores de mutación con `MUTATEXT`

De forma similar a `MODELTEXT`, el DSL `MUTATEXT` permite sobrescribir el texto por defecto que `WODEL-EDU` genera para representar cada operador de mutación aplicado. Este texto es el que se muestra como opciones de reparación en los ejercicios de selección de reparación múltiple.

El Listado 4.4 muestra un ejemplo de `MUTATEXT`. Define el texto alternativo que debe utilizarse para representar los operadores de mutación `TargetReferenceChanged`, `Attribute Changed` y `ObjectRetyped`, el segundo de los cuales sólo se aplica en caso de que se esté mutando el valor de un atributo de objetos de tipo `State`. El DSL permite configurar el texto a mostrar cuando la opción de reparación es correcta (líneas 4 y 8) y cuando no (líneas 5 y 9). Además, permite utilizar algunas variables predefinidas que contienen información sobre el operador de mutación aplicado, como `%object` que identifica el objeto mutado, o `%refName` que contiene el nombre de la referencia utilizada en el operador de mutación. Estas variables devolverán la representación textual del objeto o de la referencia especificada con `MODELTEXT`, o una representación textual por defecto si no se ha indicado ninguna. Por ejemplo, el texto definido en la línea 4 del Listado 4.4, combinado con la utilización de la definición `MODELTEXT` en el Listado 4.3, generará reparaciones como: `Change Transition a from State s0 to State s1 with new target State q2`. En caso de que no se hubiese utilizado la definición `MODELTEXT`, se obtendría la siguiente reparación: `Change Transition from s0 to s1 with new tar q2`.

```

1 metamodel "http://fa.com"
2
3 > TargetReferenceChanged:
4   Change %object from %fromObject to %toObject with new %refName %oldToObject /
5   Change %object from %fromObject to %oldToObject with new %refName %toObject
6
7 > AttributeChanged (State):
8   Change attribute %attName from %object with value %newValue to %oldValue /
9   Change attribute %attName from %object with value %oldValue to %newValue
10
11 > ObjectRetyped :
12   Retype %fromType %fromObject as %toType /
13   Retype %toType %toObject as %fromType

```

Listado 4.4 Definiendo la descripción textual de los operadores de mutación con `MUTATEXT`.

4.3. Generación de ejercicios

A continuación, esta sección ilustra los tres tipos diferentes de ejercicios generados por `WODEL-EDU`. En primer lugar, se debe especificar el programa `WODEL` que se usará para generar soluciones incorrectas a partir de soluciones correctas. A modo de ejemplo, utilizaremos `WODEL` del Listado 4.5, que crea un conjunto de mutantes a partir de los modelos semilla de autómatas situados en la carpeta `model`. El programa incluye una restricción OCL que requiere que todos los estados en los mutantes generados sean alcanzables desde el estado inicial, para así evitar la generación de autómatas trivialmente incorrectos (líneas 30-33). A partir de los mutantes generados, de la definición de los ejercicios con `EDUTEST`, y de la definición de la representación gráfica de los modelos con `MODELDRAW`, `WODEL-EDU` genera una aplicación web con una página para cada grupo de ejercicios definidos. De manera opcional, los DSLs `MODELTEXT` y `MUTATEXT` permiten personalizar el texto de las reparaciones propuestas en los ejercicios de *selección de reparación múltiple*, o en otro caso, se genera un texto por defecto coherente con la reparación. La aplicación generada para autómatas finitos puede accederse en la dirección <http://www.wodel.eu/comlan16/test.html>.

En el Listado 4.5, el primer bloque (líneas 6-11) produce un mutante para cada modelo semilla. El número de mutantes `n` que generará cada bloque se indica mediante la declaración “[`n`]” incluida a continuación de la declaración del bloque (líneas 11, 16, 22 y 26). `WODEL-EDU` utiliza estos mutantes para generar una página web con un ejercicio de *respuesta alternativa* por cada modelo semilla. Cada ejercicio muestra aleatoriamente o bien el modelo semilla (en cuyo caso es correcto), o bien el mutante (en cuyo caso es incorrecto), y los estudiantes deben responder si el autómata es correcto o no. La Figura 4.3 muestra una captura de pantalla con uno de los ejercicios de este tipo.


```

1 generate mutants in "out/" from "model/"
2 metamodel "http://fa.com"
3
4 with blocks {
5   // mutations for exercises of type alternative-response
6   alternative {
7     s0 = select one InitialState
8     s1 = select one CommonState
9     t0 = select one Transition where {src = s0}
10    modify one Transition where {tar = s1} with {swapref(tar, t0.tar)}
11  } [1]
12
13 // mutations for exercises of type multiple-diagram-choice
14 multiple {
15   modify target tar from one Transition to other State
16 } [2]
17
18 // mutations for exercises of type multiple-emendation-choice
19 incorrect_automaton {
20   modify target tar from one Transition to other State
21   retype one [CommonState, FinalState] as [CommonState, FinalState]
22 } [3]
23 incorrect_emend from incorrect_automaton repeat=no {
24   modify target tar from one Transition to other State
25   retype one [CommonState, FinalState] as [CommonState, FinalState]
26 } [6]
27 }
28
29 constraints {
30   context State connected:
31     "Set{self}→closure(s | Transition.allInstances())
32     →select(t | t.tar=s)→collect(src))
33     →exists(s | s.oclIsKindOf(InitialState))"
34 }

```

Listado 4.5 Programa WODEL utilizado para generar ejercicios de autómatas

El segundo bloque del Listado 4.5 (líneas 14-16) genera 2 mutantes a partir de cada modelo semilla. A continuación, WODEL-EDU genera una página web con ejercicios de *selección de diagrama múltiple*, donde cada modelo semilla se muestra entre sus mutantes, y los estudiantes deben identificar cuál es el correcto. Como ejemplo, la Figura 4.4 muestra uno de estos ejercicios.

Los bloques tercero y cuarto (líneas 19-26) generan los mutantes necesarios para ejercicios de tipo *selección de reparación múltiple*. Estos ejercicios muestran un autómata incorrecto junto con una lista de reparaciones expresadas de manera textual, y se pide a los estudiantes que seleccionen el subconjunto de reparaciones que podría corregir el autómata. El primer bloque (líneas 19-22) genera el autómata incorrecto, y el registro de los operadores de mutación aplicados se utiliza para generar automáticamente las opciones de reparación correctas. El operador de mutación de la línea 21 “retype one [CommonState,

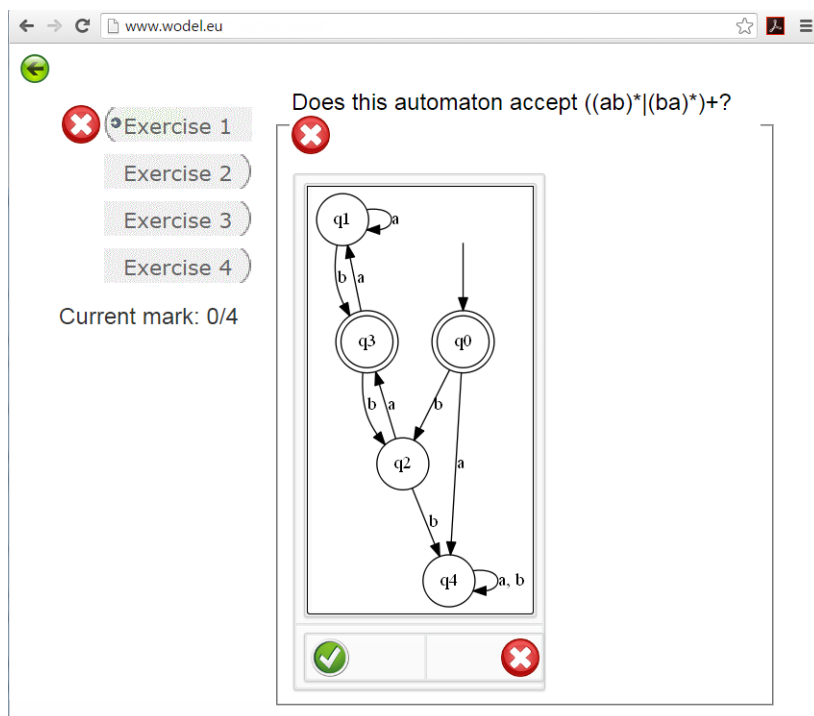


Figura 4.3 Captura de pantalla de un ejercicio de *respuesta alternativa*.

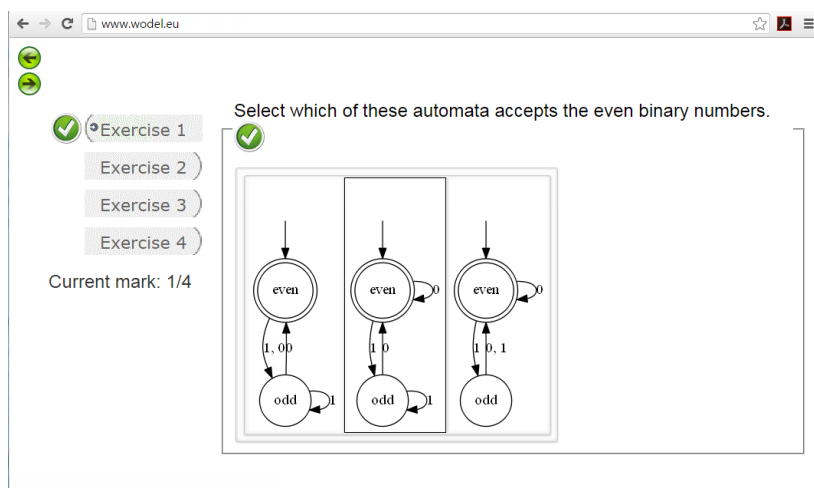


Figura 4.4 Captura de pantalla de un ejercicio de *elección de diagrama múltiple*.

`FinalState]` as `[CommonState, FinalState]`” retipa un objeto de una de las clases del listado origen a un objeto de uno de los tipos incluidos en el listado destino que es distinto de la clase original. El siguiente bloque (líneas 23-26) muta los autómatas generados en el primer bloque. Estos nuevos mutantes se descartan, y sólo se utiliza el registro de los operadores de mutación aplicados para generar las opciones de reparación incorrectas. En este ejemplo, las instrucciones de mutación en los dos bloques son las mismas: cambian el estado destino de una transición aleatoria, y después retipan un estado final a no final y viceversa. Esto tiene el efecto de generar reparaciones correctas e incorrectas, lo que incrementa la dificultad de los ejercicios. La Figura 4.5 muestra uno de los ejercicios generados, en el que deberían seleccionarse las dos últimas opciones para resolver correctamente el ejercicio.

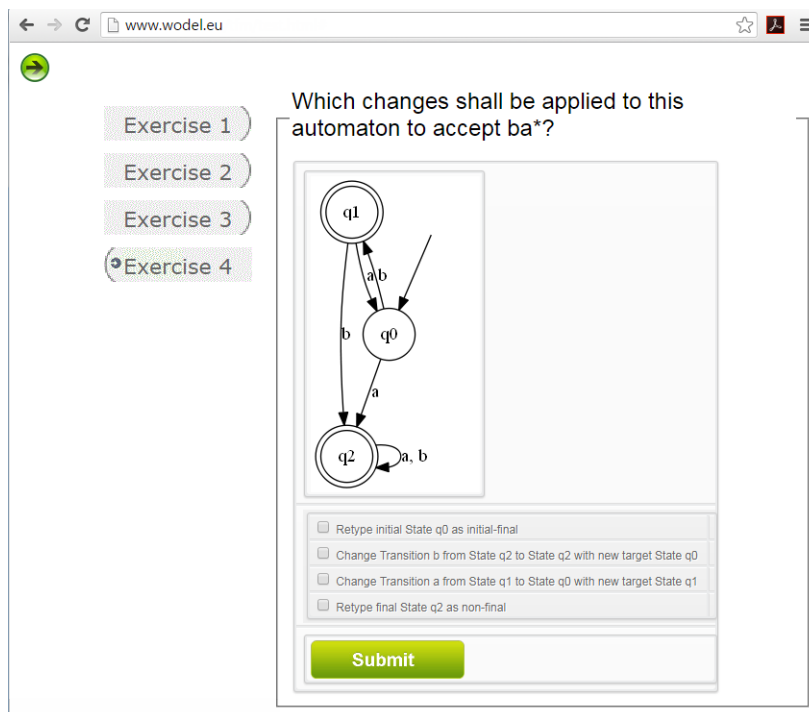


Figura 4.5 Captura de pantalla de un ejercicio de *selección de reparación múltiple*.

Entrando en detalle, el ejercicio previo se genera a partir del autómata semilla mostrado en la Figura 4.6a, que acepta el lenguaje “ba*”. Este autómata se muta de la siguiente forma: el estado q2 se transforma a final, y en la transición de q1 a q1 el estado destino pasa a ser q0. La Figura 4.6b muestra el mutante resultante, que es el que se muestra en el ejercicio. WODEL-EDU utiliza el registro de los operadores de mutación aplicados para generar el texto de las reparaciones correctas (las dos últimas en la Figura 4.5). Estas reparaciones, si se aplicaran sobre el mutante, recuperarían el modelo semilla original. En el siguiente paso, el autómata mutado es mutado nuevamente, dando como resultado el autómata en la Figura 4.6c,

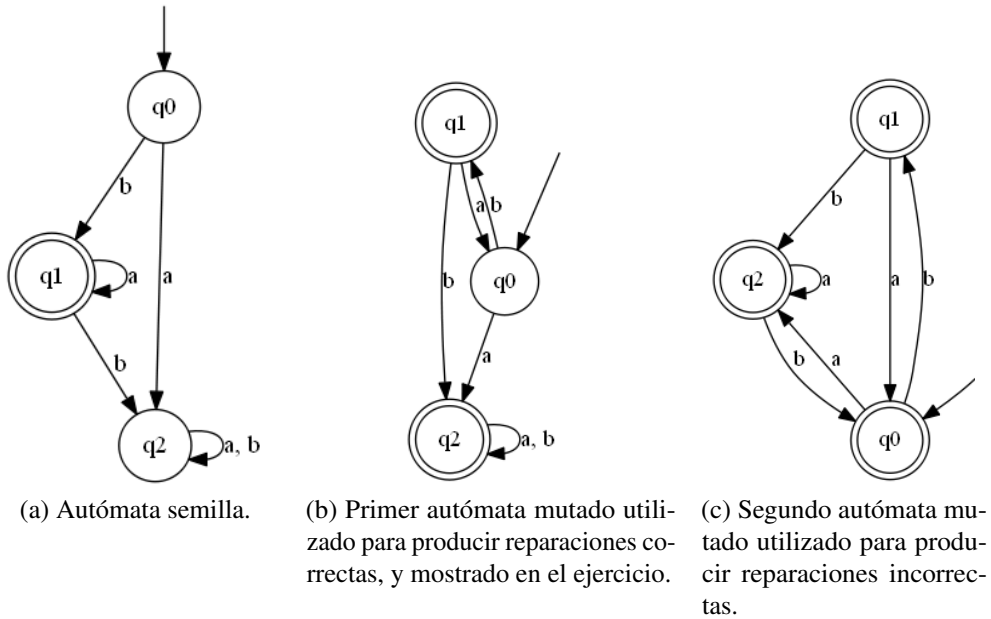


Figura 4.6 Pasos para generar el ejercicio en la Figura 4.5.

donde el estado q_0 se convierte en final, y el estado destino de la transición b desde q_2 a q_2 pasa a ser q_0 . Estos operadores de mutación se utilizan para generar dos reparaciones incorrectas en el ejercicio (las dos primeras opciones en la lista). Finalmente, *WODEL-EDU* presenta el conjunto de reparaciones correctas e incorrectas en un orden aleatorio.

WODEL-EDU puede utilizarse para crear ejercicios similares para otros dominios (p. ej., diagramas de clase) proporcionando un programa *WODEL* con los operadores de mutación de interés, indicando cómo se visualizan los elementos del modelo, y proporcionando una descripción de los ejercicios tal y como se detalla en la Sección 4.2.

4.4. Evaluación de la calidad de los ejercicios generados

En esta sección, se presentan los resultados de un caso de estudio con usuarios cuyo objetivo es medir la calidad de los ejercicios generados de forma automática con *WODEL-EDU*. Con este propósito, hemos utilizado *WODEL-EDU* para crear una aplicación web con tres páginas de ejercicios de autómatas finitos: la primera contiene ejercicios con opciones de reparación múltiple, la segunda contiene ejercicios con opciones de diagrama múltiple, y la tercera incluye ejercicios con respuesta alternativa correcta/incorrecta. Los ejercicios utilizados en la evaluación están disponibles en <http://www.wodel.eu/comlan16>.

El experimento se realizó con 10 participantes, 8 hombres y 2 mujeres, con edad comprendida entre 22 y 41 (31 años de media). Cinco son profesores de informática en la universidad,

tres son estudiantes de doctorado, uno es estudiante del grado en informática, y el último trabaja como investigador informático en una empresa. Sólo uno de ellos carecía de formación en teoría de autómatas.

Se pidió a los participantes resolver los ejercicios en la aplicación generada sin restricciones de tiempo. A continuación, debían evaluar cada página de ejercicios (es decir, cada tipo de ejercicios) con un valor entre 1 (completamente en desacuerdo) y 5 (completamente de acuerdo) con respecto a las siguientes preguntas:

- El ejercicio es fácil de entender.
- El ejercicio tiene un nivel de dificultad adecuado.
- El ejercicio es útil para aprender autómatas finitos.

De forma opcional, los participantes podían indicar la nota final obtenida en cada página de ejercicios, así como proporcionar comentarios y sugerencias.

Entre los comentarios, había varias sugerencias relacionadas con la usabilidad de la interfaz de usuario de la aplicación. En particular, varios participantes consideraron el enunciado de los ejercicios difícil de entender, y propusieron utilizar comillas alrededor de las expresiones regulares. Otro participante indicó que los ejercicios de reparación múltiple no dejaban claro que sólo debían seleccionarse las reparaciones que tenían el efecto de que el autómata aceptara las palabras del lenguaje indicado y *ninguna otra palabra*. A la vista de estos comentarios, se ha modificado el generador de código para incluir explicaciones más precisas en los ejercicios, y entrecomillar las expresiones regulares. En cualquier caso, ninguna de las sugerencias recibidas estaba relacionada con los ejercicios en sí mismos, sino con su enunciado.

Además, un participante detectó que el segundo ejercicio de la primera página contenía una reparación que no podía aplicarse en el diagrama mostrado. Inspeccionando el código, descubrimos que había un error en el generador de código. Este error, que era difícil de identificar porque sólo aparecía de forma ocasional, se ha corregido en la versión actual de WODEL-EDU.

La Figura 4.7 muestra la media de puntuaciones de los participantes para las tres páginas de ejercicios en las tres dimensiones consideradas: legibilidad, dificultad y utilidad para el aprendizaje de autómatas. En la Figura 4.7a, se puede observar que la legibilidad de los ejercicios no es mala pero tiene margen de mejora. La puntuación más baja (68 %) corresponde a la primera página que contiene los ejercicios con opciones de reparación múltiple. Las segunda y tercera páginas de ejercicios tienen una puntuación de 78 % y 76 % respectivamente. No es sorprendente que la puntuación más baja (1 de 5) fuese otorgada

por el participante sin conocimiento en teoría de autómatas. En cuanto a la dificultad de los ejercicios, la Figura 4.7b muestra que la primera página de ejercicios también se considera la más difícil, con un porcentaje del 82 %. Las otras dos páginas tienen puntuaciones de 89 % (elección de diagrama múltiple) y 86 % (respuesta alternativa). Con respecto a la última dimensión, la Figura 4.7c muestra que los participantes consideraron los ejercicios muy útiles para aprender autómatas, con la puntuación de 88 % para la página con ejercicios de respuesta alternativa, y de 94 % para las páginas con opciones múltiples.

La Figura 4.8 muestra la media de la nota final obtenida por los participantes en las tres páginas de ejercicios, aunque sólo el 60 % de los participantes proporcionaron su nota en los cuestionarios. De media, la nota final obtenida está sobre el 60 % en las páginas segunda y tercera, y en el 50 % en la primera página. Dos participantes obtuvieron la nota máxima en la primera página, un participante obtuvo la nota máxima en la segunda página, y dos participantes obtuvieron la nota máxima en la tercera página. Ningún participante obtuvo la nota máxima en todas las páginas.

Se pueden extraer algunas conclusiones interesantes de esta evaluación. De acuerdo con los resultados, los ejercicios con opciones de reparación múltiple son difíciles de entender. Probablemente esto se deba a que puede resultar difícil representar mentalmente el resultado de aplicar las reparaciones al autómata. Una solución a este problema podría ser mostrar el autómata resultante tras aplicar el conjunto de reparaciones seleccionadas. Otra opción podría ser hacer el ejercicio interactivo, permitiendo a los usuarios realizar cambios en el autómata presentado para construir la solución correcta. Por otro lado, los participantes consideraron que los ejercicios tienen un nivel adecuado de dificultad, y que son muy útiles para aprender autómatas. Una última cuestión es la necesidad de proporcionar una descripción más precisa a los ejercicios, o incluir un tutorial para entender los ejercicios. Como prueba de esto, el participante que no tenía práctica en teoría de autómatas dio la puntuación más baja a la legibilidad de los tres tipos de ejercicio. Todas estas ideas son muy útiles para mejorar WODEL-EDU.

También hay que mencionar algunos aspectos que pueden afectar a la validez de esta evaluación. Primero, el orden en el que se muestran los ejercicios a los participantes puede haber influido en su percepción de la dificultad de los ejercicios. Por ejemplo, la primera página de ejercicios que contenía opciones de reparación múltiple es la más difícil, y esto puede haber afectado la complejidad percibida de la segunda página de ejercicios. Otro aspecto a tener en cuenta es el hecho de que todos los ejercicios son sobre autómatas finitos; por tanto, las conclusiones de este caso de estudio podrían ser diferentes si se consideran ejercicios en otros dominios. Además, la evaluación sólo considera los ejercicios generados de forma automática. Para complementar estos resultados, se debería realizar otro estudio

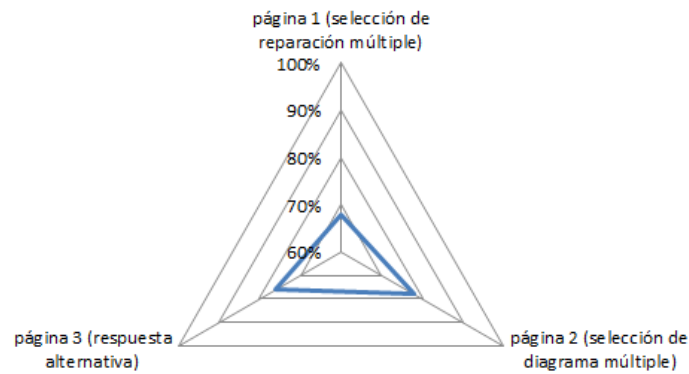
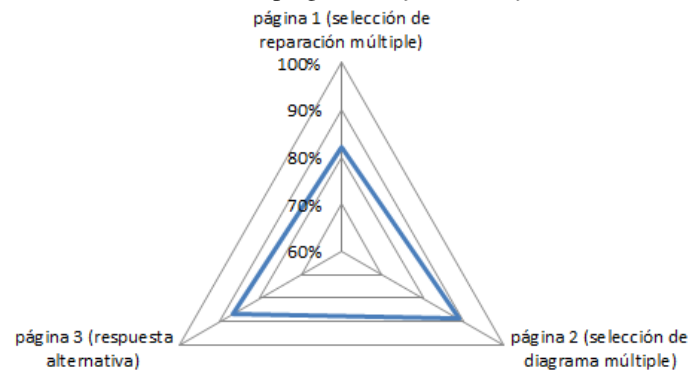
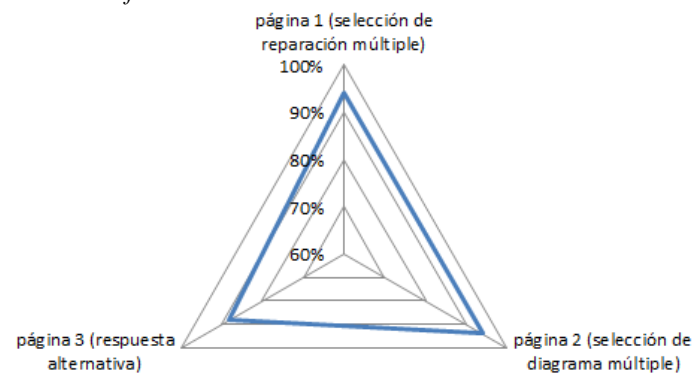
(a) Puntuación media a la pregunta *el ejercicio es fácil de entender*.(b) Puntuación media a la pregunta *el ejercicio tiene un nivel adecuado de dificultad*.(c) Puntuación media a la pregunta *el ejercicio es útil para aprender autómatas*.

Figura 4.7 Puntuación media de las páginas de ejercicios consideradas en la evaluación.

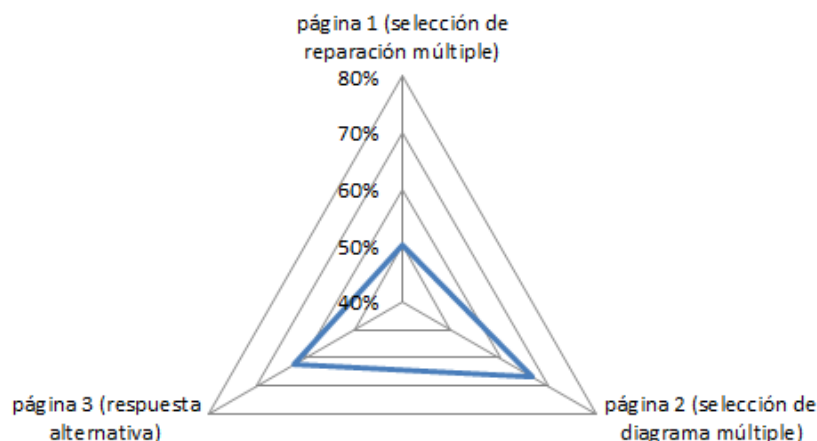


Figura 4.8 Nota media de los participantes en las páginas de ejercicios.

donde los participantes resolvieran tanto ejercicios generados de forma automática como otros diseñados manualmente. Esto permitiría comparar los resultados obtenidos en ambos casos, así como evaluar si los participantes son capaces de distinguir qué ejercicios son generados y cuáles son creados por un humano. Respecto a la generalidad de los resultados, los participantes en este estudio no estaban estudiando cursos de teoría de autómatas en ese momento o recientemente, y uno de ellos no tenía práctica previa en esta materia. Por tanto, los resultados pueden ser diferentes en el caso de participantes que estén realizando cursos sobre autómatas. El experimento se realizó con sólo 10 participantes, lo que es poco significativo, y por tanto, se debería realizar otro experimento con más participantes, así como evaluar la herramienta desde la perspectiva del profesor que está a cargo del diseño de los ejercicios.

Capítulo 5

Wodel-Test: Pruebas de Mutación Independientes del Lenguaje

En este capítulo se presenta `WODEL-TEST`, el post-procesador de `WODEL` para la creación de entornos de pruebas de mutación independientes del lenguaje. `WODEL-TEST` puede usarse para crear herramientas de pruebas de mutación para cualquier lenguaje de programación o modelado definido mediante un meta-modelo, o para el que se especifique cómo obtener una representación en forma de modelo de los programas (y viceversa). La Sección 5.1 muestra la visión general del enfoque propuesto para sintetizar herramientas de pruebas de mutación para lenguajes arbitrarios. Posteriormente, la Sección 5.2 presenta la arquitectura y la funcionalidad de las herramientas de pruebas de mutación que se sintetizan. Por último, para demostrar la utilidad de la propuesta, la Sección 5.3 presenta una evaluación de las dos herramientas de pruebas de mutación que se han generado con `WODEL-TEST`, la primera para programas Java y la segunda para programas ATL.

5.1. Visión general

La Figura 5.1 muestra un esquema de nuestro enfoque para especificar herramientas de pruebas de mutación. En el proceso se distinguen dos roles: el *creador de la herramienta de pruebas de mutación* y el *ingeniero de pruebas*. El primero proporciona una especificación de la herramienta de pruebas de mutación a partir de la cual ésta es sintetizada. El segundo utiliza la herramienta generada para realizar las pruebas de mutación. A continuación, se detallan las actividades realizadas por cada rol.

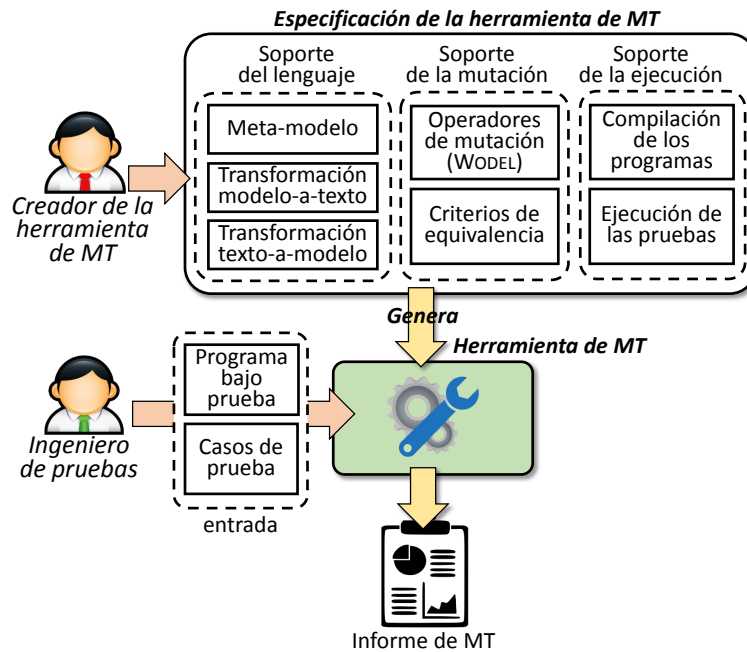


Figura 5.1 Creación y utilización de una herramienta de pruebas de mutación con WODEL-TEST.

Durante la primera fase tiene lugar la creación de la herramienta de pruebas de mutación. Con este objetivo, el creador de la herramienta de pruebas de mutación tiene que proporcionar los siguientes grupos de artefactos:

- especificación del *soporte del lenguaje* para el que se está creando la herramienta de pruebas de mutación. Ésta consiste en el meta-modelo del lenguaje objetivo, y en caso de ser necesario, una especificación de cómo convertir los programas del lenguaje en instancias de este meta-modelo, y viceversa. De este modo, si los programas del lenguaje son ficheros de código fuente en texto (por ejemplo, programas Java), se requiere que el creador de la herramienta proporcione una transformación texto-a-modelo que cree un modelo a partir de los ficheros de texto, y una transformación modelo-a-texto para serializar el modelo a texto tras aplicar los operadores de mutación. Si el lenguaje no es textual sino gráfico (por ejemplo, diagramas de clases), se requiere que el creador de la herramienta proporcione las transformaciones para obtener el modelo en sintaxis abstracta a partir de su representación en sintaxis gráfica, y viceversa. Si se está trabajando con modelos sin sintaxis concreta (p. ej., modelos puros definidos con el Eclipse Modeling Framework (EMF) [118], que sólo contienen información de la sintaxis abstracta), estas transformaciones no son necesarias.

En el caso de que estos artefactos (el meta-modelo y las transformaciones texto-a-modelo y modelo-a-texto) no existan será necesario crearlos. En el caso de querer aplicar `WODEL-TEST` a un lenguaje de programación, cabe destacar que cada vez es más fácil encontrar artefactos de modelado disponibles para los lenguajes de programación más establecidos, como Java, COBOL, C++, C o SmallTalk [26, 47, 80], ya que MDE se está empezando a utilizar de manera frecuente en proyectos de reingeniería software [1]. La probabilidad de encontrar y reutilizar estos artefactos para lenguajes de modelado (p. ej., UML, BPMN) es incluso mayor.

- especificación del *soporte de la mutación* de la herramienta de pruebas de mutación. Ésta consiste en la codificación en `WODEL` de los operadores de mutación a utilizar para crear mutantes de los programas, y en la definición de criterios de equivalencia sintáctica y semántica que se utilizarán para detectar mutantes equivalentes.
- especificación del *soporte de ejecución* de la herramienta de pruebas de mutación, consistente en la definición del proceso de compilación del programa bajo prueba y los mutantes generados, así como la ejecución de los conjuntos de pruebas.

La compilación *Just-in-time* es una forma de ejecutar el código mediante una compilación en tiempo de ejecución [13]. Ésta consiste en una traducción de código fuente, o de forma más habitual de bytecode, a código máquina. Por tanto, este tipo de compilación es recomendado debido a su eficiencia.

Existen algunos lenguajes de modelado (p. ej., autómatas) para los que la compilación no es necesaria. En estos casos, tan solo es necesario especificar cómo se ejecutan los programas y los mutantes.

`WODEL-TEST` recibe estas tres especificaciones como entrada y sintetiza automáticamente una herramienta de pruebas de mutación orientada al lenguaje objetivo. El ingeniero de pruebas podrá utilizar esta herramienta proporcionando el programa bajo prueba y un conjunto de casos de prueba. La herramienta proporciona como salida una serie de métricas descriptivas del proceso de pruebas de mutación entre las que se incluye el *mutation score*. La Sección 5.2 presenta las funcionalidades que ofrecen las herramientas de pruebas de mutación generadas.

Como ejemplo, la Figura 5.2 muestra el esquema de funcionamiento interno de la herramienta de pruebas de mutación generada para el lenguaje Java. Dicha herramienta se presentará en detalle en la sección 5.3.1.1. En primer lugar, la herramienta transforma el programa bajo prueba a un modelo conforme al meta-modelo Java que proporciona `MoDisco` [26] utilizando la transformación texto-a-modelo especificada. A continuación, el modelo se muta aplicando los operadores de mutación definidos, y cada mutante se serializa

de nuevo a formato textual utilizando la transformación modelo-a-texto. Finalmente, se compilan el programa original y los programas mutantes en formato textual y se les aplican los casos de prueba.

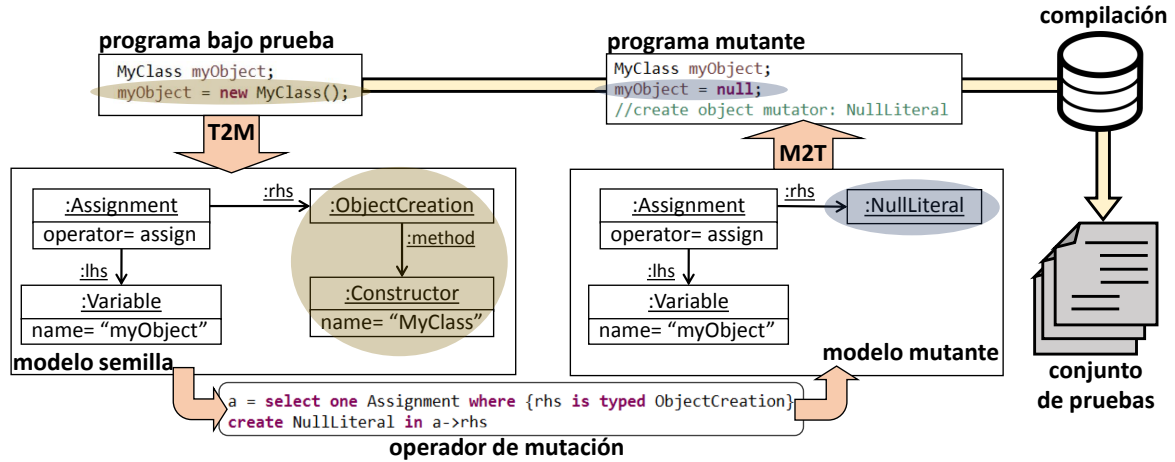


Figura 5.2 Esquema del funcionamiento de la herramienta de pruebas de mutación para Java.

El esquema de funcionamiento definido de este modo es similar a los enfoques de pruebas de mutación que funcionan a nivel del árbol de sintaxis abstracta (AST). Sin embargo, los modelos no necesitan estar en forma de árbol (los modelos son grafos) y su sintaxis concreta no tiene por qué ser textual. La siguiente sección presenta la arquitectura de la herramienta proporcionada.

5.2. Herramienta proporcionada

La Figura 5.3 muestra la arquitectura modular de WODEL-TEST basada en componentes que consiste en un conjunto de plugins de Eclipse. La tecnología de modelado subyacente es Eclipse Modeling Framework (EMF) [118], y extiende el motor de la herramienta WODEL con funcionalidades específicas para las pruebas de mutación (etiqueta 1). WODEL y WODEL-TEST están disponibles en <http://gomezabajo.github.io/Wodel/>. El sitio web incluye instrucciones de instalación, ejemplos, demos, y el código fuente.

Como se explicó en el capítulo 3, WODEL proporciona puntos de extensión para definir criterios sintácticos y semánticos, que WODEL-TEST utiliza para descartar mutantes equivalentes en el cálculo del mutation score. A su vez, WODEL-TEST (etiqueta 2) está diseñado como una extensión de post-procesado de WODEL.

Si bien definir un criterio de equivalencia es opcional al crear una nueva herramienta de pruebas de mutación, WODEL-TEST requiere necesariamente que el creador de la

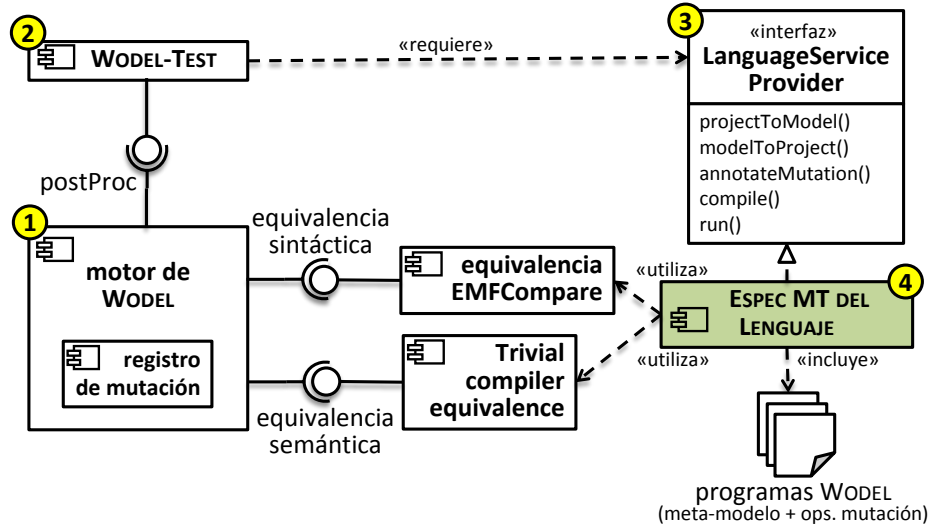


Figura 5.3 Arquitectura de WODEL-TEST.

herramienta de pruebas de mutación proporcione uno o más programas WODEL con la definición de los operadores de mutación, así como implementar una interfaz llamada `LanguageServiceProvider` (etiqueta 3). Esta interfaz presenta los siguientes métodos:

- **projectToModel**: Este método tiene como función convertir el artefacto a mutar en un modelo EMF (p. ej., realiza una transformación texto-a-modelo).
- **modelToProject**: Este método se encarga de convertir el modelo que representa el programa mutado en un artefacto del dominio (es decir, realiza una transformación modelo-a-texto).
- **annotateMutation**: Este método se puede utilizar para insertar comentarios en el código mutado, describiendo las mutaciones realizadas. WODEL-TEST utiliza el registro de operadores de mutación aplicados de WODEL para identificar las mutaciones aplicadas. De esta forma, el creador de la herramienta de pruebas de mutación sólo tiene que identificar la clase del meta-modelo que corresponde a la representación de comentarios en el código, y añadir una instancia del mismo en los modelos mutantes incluyendo el texto descriptivo de la mutación aplicada.
- **compile**: Este método debe incluir código para compilar el programa original y los artefactos obtenidos de las mutaciones aplicadas.
- **run**: Este método debe codificar cómo ejecutar los programas contra el conjunto de pruebas.

Una vez implementada la interfaz y definidos el meta-modelo del lenguaje objetivo, los operadores de mutación y (opcionalmente) los criterios de equivalencia, se genera la herramienta de pruebas de mutación para el lenguaje.

Las herramientas sintetizadas por WODEL-TEST permiten aplicar pruebas de mutación sobre programas codificados en el lenguaje seleccionado y los conjuntos de prueba correspondientes. El ingeniero de pruebas puede seleccionar los operadores de mutación que desea incluir en el proceso de pruebas de mutación, entre los definidos por el creador de la herramienta. Este proceso se realiza utilizando la ventana de preferencias que se genera en la creación de las herramientas de pruebas de mutación. La Figura 5.4 corresponde a la creada en la herramienta de pruebas de mutación para Java que se presentará en la Sección 5.3.1.1. En esta ventana, los operadores de mutación se muestran agrupados por categoría.

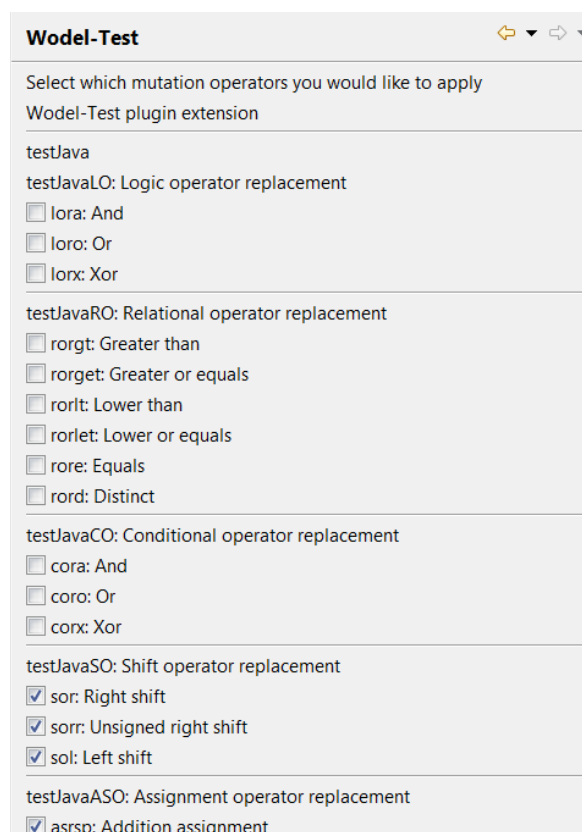


Figura 5.4 Selección de operadores de mutación en la ventana de preferencias (extracto).

Los resultados del proceso de pruebas de mutación pueden visualizarse en cuatro vistas de Eclipse distintas. Las Figuras 5.5 a 5.7 muestran estas vistas para la herramienta de pruebas de mutación creada para Java. En concreto, la Figura 5.5 muestra la vista global de resultados tras aplicar el proceso de pruebas de mutación a un proyecto Java. La etiqueta 1 corresponde al explorador de proyectos de Eclipse, que contiene una carpeta `src` con el proyecto Java

bajo prueba, y una carpeta para cada uno de los operadores de mutación que se han aplicado que contienen los mutantes correspondientes. La etiqueta 2 muestra un mutante, en el que un comentario debajo del código mutado describe la mutación aplicada. En este caso, el operador “%” se ha reemplazado por “/”. La etiqueta 3 muestra el conjunto de pruebas. Tanto el conjunto de pruebas como el proyecto Java bajo prueba son proyectos estándar de Eclipse. La etiqueta 4 muestra la vista global de resultados que incluye el mutation score, el tiempo de ejecución del proceso de pruebas de mutación, el número de operadores de mutación que se han aplicado y el número de los operadores que no se han aplicado, el número de mutantes muertos, equivalentes y vivos así como el número de pruebas superadas y no superadas. Esta información se muestra utilizando gráficos de barras que presentan los porcentajes correspondientes.

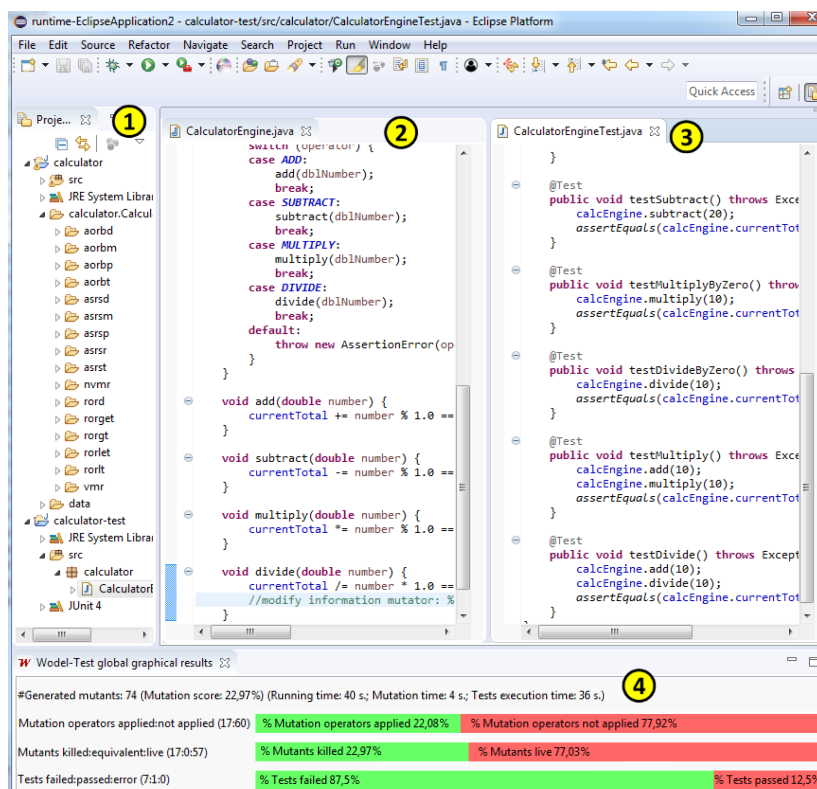


Figura 5.5 Vista global de los resultados de pruebas de mutación.

La herramienta generada ofrece dos vistas detalladas de los resultados que se centran en los mutantes y las pruebas. La Figura 5.6a muestra la vista correspondiente a los mutantes, con los resultados de las pruebas superadas y no superadas por cada mutante. Las pruebas no superadas, es decir, aquellas que detectan las mutaciones introducidas, se muestran en verde, y las pruebas superadas se muestran en rojo. Los resultados se pueden filtrar para mostrar sólo las pruebas superadas o las no superadas. Para cada mutante, la vista incluye una

descripción de los operadores de mutación aplicados (última columna). También se indica si el mutante es equivalente o no de acuerdo con los criterios de equivalencia sintáctica y semántica especificados por el creador de la herramienta de pruebas de mutación (primera columna). Esta vista también permite que el ingeniero de pruebas marque cada mutante vivo como equivalente. En ese caso, la herramienta recalcula automáticamente las estadísticas de la vista global teniendo en cuenta la variación del número de mutantes equivalentes. La herramienta también permite mostrar los mutantes en el editor de Eclipse, haciendo doble-clic en la ruta que se muestra.

La Figura 5.6b muestra la segunda vista detallada, que considera los mutantes muertos y vivos tras la ejecución de los casos de prueba. De la misma forma que en la vista por mutantes, el ingeniero de pruebas puede editar los mutantes haciendo doble-clic en la ruta que se muestra, y filtrar sólo los mutantes vivos o muertos.

Finalmente, se proporciona una vista con detalles del proceso de generación de los mutantes, que muestra los mutantes que cada operador de mutación ha generado. La Figura 5.7 muestra esta vista, en la que es posible filtrar los operadores de mutación aplicados y los no aplicados.

5.3. Evaluación

En esta sección, evaluamos nuestro enfoque desde el punto de vista de los dos perfiles de usuario involucrados en el proceso: el ingeniero de pruebas y el creador de la herramienta de pruebas de mutación.

En cuanto al ingeniero de pruebas, la Sección 5.3.1 presenta un experimento para determinar la utilidad de las herramientas de pruebas de mutación que se generan utilizando nuestra propuesta. Con este objetivo, se detalla la construcción de una herramienta de MT para Java creada con `WODEL-TEST`, y se compara con otras herramientas existentes de pruebas de mutación para Java. Esta comparativa tiene como objetivo responder la siguiente pregunta de investigación:

RQ1: *¿Permite `WODEL-TEST` crear herramientas de pruebas de mutación con capacidades similares a las herramientas de pruebas de mutación creadas manualmente?*

Para evaluar nuestro enfoque desde el punto de vista del creador de la herramienta de pruebas de mutación, la Sección 5.3.2 presenta un caso de estudio que detalla la construcción de una herramienta de pruebas de mutación para el *Atlas transformation language* (ATL) [65].

W Model-Test mutant results

Filter

All

Failed

Passed

Results

Equivalent	Package/class/mutant	#Exe...	#Fail...	#Pass...	Applied mutations/Failed test message
▲	calculator	264	16	248	
▲	CalculatorEngine	264	16	248	
▶	/calculator/src/calculator/C...	8		8	
▶	/calculator/aorbd/Output0...	8		8	modify information mutator: % replaced by /
▶	/calculator/aorbd/Output1...	8		8	modify information mutator: % replaced by /
▶	/calculator/aorbm/Output0...	8		8	modify information mutator: % replaced by -
▶	/calculator/aorbm/Output1...	8		8	modify information mutator: % replaced by -
▶	/calculator/aorbp/Output0...	8		8	modify information mutator: % replaced by +
▶	/calculator/aorbp/Output1...	8		8	modify information mutator: % replaced by +
▶	/calculator/aorbt/Output0/...	8		8	modify information mutator: % replaced by *
▶	/calculator/aorbt/Output1/...	8		8	modify information mutator: % replaced by *
▲	/calculator/asrsd/Output0/...	8	3	5	modify information mutator: += replaced by /=
	testSubtract	1		1	
	testDivide	1	1		expected:<0.0> but was:<20.0>
	testMultiplyByZero	1		1	
	testAdd	1	1		expected:<0.0> but was:<1.0>
	testMultiply	1	1		expected:<0.0> but was:<100.0>
	testGetTotalStringInt	1		1	
	testEqual	1		1	
	testDivideByZero	1		1	
▶	/calculator/asrsm/Output1/...	8	1	7	modify information mutator: *= replaced by /=

(a) Pruebas superadas y no superadas por cada mutante.

W Model-Test test results

Filter

All

Killed

Alive

Results

Test suite/Test case	#Killed mutants/Message	Applied mutations
▲ /calculator-test/src/calculator/Cal	10	
▶ testSubtract	2	
▶ testDivide	4	
▶ testMultiplyByZero	2	
▶ testAdd	1	
▶ testMultiply	5	
▶ testGetTotalStringInt		
▶ testEqual		
▲ testDivideByZero	2	
/calculator/src/calculator/C		
/calculator/aorbd/Output0/		modify information mutator: % replaced by /
/calculator/aorbd/Output1/		modify information mutator: % replaced by /
/calculator/aorbm/Output0/		modify information mutator: % replaced by -
/calculator/aorbm/Output1/		modify information mutator: % replaced by -
/calculator/aorbp/Output0/		modify information mutator: % replaced by +
/calculator/aorbp/Output1/		modify information mutator: % replaced by +
/calculator/aorbt/Output0/s		modify information mutator: % replaced by *
/calculator/aorbt/Output1/s		modify information mutator: % replaced by *
/calculator/asrsd/Output0/s		modify information mutator: += replaced by /=
/calculator/asrsd/Output1/s		modify information mutator: *= replaced by /=
/calculator/asrsm/Output0/s		modify information mutator: *= replaced by -=
/calculator/asrsm/Output1/s expected:<0.0> but was:<1.0>		modify information mutator: /= replaced by +=
/calculator/asrsp/Output0/s expected:<20.0> but was:<1.0>		modify information mutator: /= replaced by +=

(b) Mutantes muertos y vivos por cada prueba.

Figura 5.6 Vistas detalladas de los resultados del proceso de pruebas de mutación.

ATL es un lenguaje basado en reglas utilizado ampliamente en MDE para describir transformaciones modelo a modelo. Hemos seleccionado este lenguaje como caso de estudio porque tiene algunos aspectos (p. ej., la necesidad de acceder a varios artefactos durante el proceso de mutación) que permiten mostrar el gran potencial de WODEL-TEST. El objetivo con este caso de estudio es responder a la siguiente pregunta de investigación:

Filter	Mutation operator/description	Generated mutants/paths
All	testJavaLO/Logic operator replacement	0
Applied	testJavaRO/Relational operator replacement	10
Not applied	testJavaCO/Conditional operator replacement	0
	testJavaSO/Shift operator replacement	0
	testJavaASO/Assignment operator replacement	10
	asrsp/Addition assignment	2
	/calculator/asrsp/Output0...	
	/calculator/asrsp/Output1...	
	asrsm/Subtraction assignment	2
	asrst/Times assignment	2
	asrds/Division assignment	2
	asrsm/Modulus assignment	2

Figura 5.7 Vista con los mutantes generados por cada operador de mutación.

RQ2: ¿Cómo de efectivo es WODEL-TEST para especificar herramientas de pruebas de mutación?

5.3.1. Comparativa de la herramienta de pruebas de mutación generada para Java con herramientas hechas manualmente

En esta sección, el objetivo es evaluar si las funcionalidades de las herramientas de pruebas de mutación generadas con nuestro enfoque son comparables con aquellas proporcionadas por herramientas de pruebas de mutación desarrolladas manualmente. Con este objetivo, presentamos la herramienta de pruebas de mutación para Java que hemos creado utilizando WODEL-TEST, y la comparamos con cuatro herramientas representativas de pruebas de mutación desarrolladas para Java: Major¹ [66], Javalanche² [108], PITest³ [31] y LittleDarwin⁴ [97]. Estas herramientas se han utilizado previamente en la literatura para realizar comparaciones de herramientas de pruebas de mutación [72, 84].

Nuestra evaluación se estructura en cuatro partes. En primer lugar, se presenta la herramienta de pruebas de mutación para Java creada con WODEL-TEST (Sección 5.3.1.1). En segundo lugar, se comparan las características de las herramientas existentes de pruebas de mutación para Java con la creada con WODEL-TEST (Sección 5.3.1.2); a continuación, se discuten las capacidades de extensibilidad y la posibilidad de configurar los operadores de mutación en las distintas herramientas (Sección 5.3.1.3); por último, se analiza la eficiencia de las herramientas de pruebas de mutación (Sección 5.3.1.4). La sección concluye respondiendo a la pregunta RQ1 y comentando los aspectos que pueden afectar a la validez de los resultados de esta evaluación (Sección 5.3.1.5).

¹<http://mutation-testing.org/>

²<http://javalanche.org/>, <https://github.com/david-schuler/javalanche>

³<http://pitest.org/>, <https://github.com/hcoles/pitest>

⁴<http://littledarwin.parsai.net/>, <https://github.com/aliparsai/LittleDarwin>

5.3.1.1. Definición de la herramienta de pruebas de mutación para Java

Como ejemplo, vamos a utilizar un pequeño subconjunto de Java, para el que crearemos una herramienta de pruebas de mutación utilizando nuestro enfoque. La Figura 5.8 muestra un fragmento del meta-modelo para el lenguaje, obtenido a partir de la herramienta de modernización basada en modelos MoDisco [26]. Este meta-modelo contiene los elementos para representar asignaciones Java, variables, expresiones binarias, algunos literales (números, cadenas y null), y llamadas a constructores.

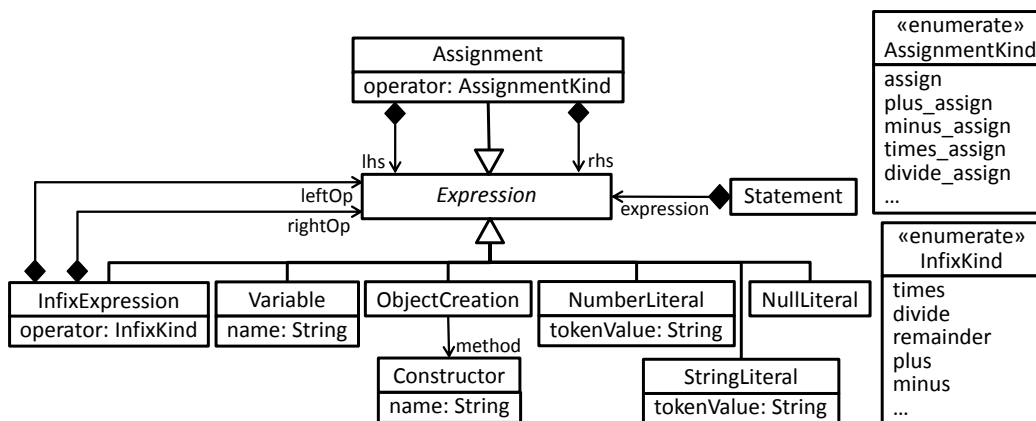


Figura 5.8 Subconjunto del meta-modelo de Java (obtenido de MoDisco [26]).

Como ejemplo, el Listado 5.1 muestra un programa WODEL sencillo, que define el operador de mutación (CIR) [78] que reemplaza una llamada a un constructor por null. La Figura 5.2 ilustra una aplicación de este operador de mutación a un modelo.

```

1 generate 2 mutants in "out/" from "model/"
2 metamodel "mm/java.ecore"
3 description "Programa Wodel sencillo"
4
5 with blocks {
6   CIR "Replace a call to a constructor by null" {
7     a = select one Assignment where { rhs is typed ObjectCreation }
8     create NullLiteral in a->rhs
9   }
}

```

Listado 5.1 Programa WODEL que define el operador de mutación CIR.

Las líneas 6–9 definen el operador de mutación. La primera instrucción (línea 7) almacena en la variable *a* un objeto aleatorio de tipo *Assignment* cuya referencia *rhs* es un objeto de tipo *ObjectCreation*. La segunda instrucción (línea 8) crea un nuevo objeto *NullLiteral* en la referencia *rhs* de la variable *a*. Este programa genera mutantes en los que se ha reemplazado la creación de un objeto en la parte derecha de una asignación Java por *null*. El

Anexo G presenta el listado completo de operadores de mutación para el meta-modelo de Java de MoDisco implementados en Wodel.

En esta herramienta de pruebas de mutación para Java, se ha implementado la técnica *trivial compiler equivalence* [71] que compara el bytecode del programa original con el bytecode de los mutantes para detectar equivalencias. La definición de un criterio de equivalencia es opcional al crear una nueva herramienta de pruebas de mutación.

En este ejemplo, se utiliza MoDisco para convertir el proyecto Java en un modelo, y se proporciona un asistente para seleccionar las clases a incluir en el proceso de pruebas de mutación. Esto último posibilita la optimización de la transformación texto-a-modelo en caso de proyectos Java grandes. La serialización del modelo generado a un proyecto Java también se lleva a cabo utilizando MoDisco. Por último, utilizamos el framework JUnit para las pruebas unitarias e invocar dinámicamente a los casos de pruebas proporcionados.

5.3.1.2. Características generales

A continuación, se analizan las características generales de las herramientas de pruebas de mutación consideradas, que se resumen en la Tabla 5.1. La comparativa se basa en tres criterios: entrada del proceso de pruebas de mutación, proceso de generación de los mutantes, e información proporcionada.

Respecto al primer criterio, todas las herramientas analizadas se ejecutan desde línea de comandos mientras que Wodel-Test, que está integrado en Eclipse, tiene una interfaz gráfica de usuario. Todas las herramientas permiten realizar pruebas de mutación sobre proyectos Java, aunque en el caso de Major y Wodel-Test es posible reducir el ámbito de aplicación a clases específicas, lo que resulta útil para reducir el proceso de pruebas de mutación en caso de proyectos grandes. Todas las herramientas dan soporte a conjuntos de pruebas especificados con JUnit 4. En el caso de Wodel-Test se ha implementado soporte para JUnit 4, aunque también se permite extender la herramienta de pruebas de mutación con las últimas versiones de JUnit (como la versión actual de JUnit⁵) y otros entornos de pruebas de unidad.

En cuanto a la generación de mutantes, cada herramienta proporciona su propio conjunto predefinido de operadores de mutación para Java, y sólo Major y Wodel-Test permiten la extensión de este conjunto utilizando DSLs (Major-Mml y Wodel respectivamente). Compararemos estos DSLs en la Sección 5.3.1.3. Major, Javalanche y PITest aplican las mutaciones a nivel del bytecode, lo que puede ser más eficiente, pero con el inconveniente de no proporcionar al ingeniero de pruebas el código de los mutantes. En cambio, LittleDarwin y Wodel-Test aplican las mutaciones al AST y una representación basada en modelos del

⁵<https://junit.org/junit5/>

	Major	Javalanche	PITest	LittleDarwin	WODEL-TEST/Java
Última actualización	2018	2011	2018	2018	2020
LOC	No disponible	15 638	15 804	14 746	305+framework

Entrada del proceso de pruebas de mutación

UI	Línea de comandos	Línea de comandos	Línea de comandos	Línea de comandos	Plugin de Eclipse
Ámbito	proyecto/clase	proyecto	proyecto	proyecto	proyecto/clase
Conjunto de pruebas	JUNIT 4	JUNIT 4	JUNIT 4	JUNIT 4	JUNIT 4 (configurable)

Proceso de generación de los mutantes

N. de operadores	30 (por defecto)	19	40	28	77 (por defecto)
Extensibilidad de ops.	Sí (DSL)	No	No	No	Sí (DSL)
Artefacto mutado	Bytecode	Bytecode	Bytecode	AST	Modelo
Código del mutante	No	No	No	Sí	Sí
Filtrado de mutantes	Sí	Sí	Sí	No	No
Detección de mutantes equivalentes	No	Sí (inv. dinámicas)	No	No	Sí (TCE)

Informes

Tipo de informe	CSV	HTML	HTML	HTML	Vistas interactivas
N. de mutantes	✓	✓	✓	✓	✓
Mutation score	✓	✓	✓	✓	✓
Mutantes muertos/vivos	Número	Número	Número	Número	Número, lista
Cobertura de operadores	Número	Número	Número	✗	Número, lista
Mutantes por clase	✗	✗	✗	✓	✓
Pruebas por mutante	✗	✗	✗	✗	✓
Mutantes por prueba	✗	✗	✗	✗	✓

Tabla 5.1 Comparativa de herramientas de pruebas de mutación para Java.

código fuente, respectivamente, pudiendo por tanto proporcionar el código de los mutantes para su inspección. Además, para mejorar el rendimiento, Major, Javalanche y PITest realizan un pre-filtrado de mutantes basado en la cobertura de instrucciones, dado que ninguna prueba puede detectar un mutante que no puede alcanzarse y ejecutarse. Para evitar evaluar estos mutantes no cubiertos, se recoge información condicional de la cobertura de los mutantes en tiempo de ejecución [66]. Otra vía para mejorar el coste de la ejecución del proceso de pruebas de mutación es la detección de mutantes equivalentes y su exclusión del proceso de pruebas de mutación. Este mecanismo sólo está soportado por Javalanche y WODEL-TEST. Javalanche detecta mutantes equivalentes evaluando el impacto de las mutaciones en invariantes dinámicas. En el caso de WODEL-TEST, los mutantes Java equivalentes se detectan utilizando la técnica *trivial compiler equivalence* (TCE) [71], pero se pueden aplicar otros criterios implementando un punto de extensión (ver Figura 3.13).

Con respecto a la salida del proceso de pruebas de mutación, todas las herramientas generan informes de los resultados basados en ficheros, mientras WODEL-TEST también puede visualizar estos resultados en vistas de Eclipse interactivas. En cuanto a la información proporcionada, todas las herramientas indican el número de mutantes generados, el mutation score, y el número de mutantes muertos y vivos. WODEL-TEST también proporciona una vista con la lista de mutantes vivos y muertos, y es posible navegar a ellos (ver Figura 5.6b). Todas

las herramientas excepto LittleDarwin informan del número de operadores de mutación cubiertos (es decir, los operadores que producen un conjunto de mutantes no vacío); WODEL-TEST también informa de la lista de operadores cubiertos y los mutantes que genera cada prueba (ver Figura 5.7). Tanto LittleDarwin como WODEL-TEST informan de los mutantes que genera cada clase, pero sólo WODEL-TEST proporciona información sobre las pruebas que supera/no supera cada mutante (ver Figura 5.6a), y los mutantes que mata cada prueba (ver Figura 5.6b).

5.3.1.3. Extensibilidad de los operadores de mutación

Entre las herramientas analizadas, sólo Major y WODEL-TEST permiten extender el conjunto predefinido de operadores de mutación, utilizando DSLs dedicados en ambos casos. Los Listados 5.2 y 5.3 ilustran los DSLs de ambas herramientas basándose en el mismo ejemplo.

Major proporciona un DSL (en realidad es un lenguaje de script) llamado Major-Mml para crear operadores de mutación y configurar el proceso de mutación. Major-Mml proporciona 8 primitivas de mutación, 7 de las cuales permiten reemplazar operadores y literales, y 1 para eliminar instrucciones. Estas primitivas son las siguientes: Arithmetic Operator Replacement (AOR), Logic Operator Replacement (LOR), Shift Operator Replacement (SOR), Conditional Operator Replacement (COR), Relational Operator Replacement (ROR), Literal Value Replacement (LVR), Operator Replacement Unary (ORU), y SStatement Deletion (STD). Estas primitivas pueden aplicarse a un paquete, clase o método específico. Además, es posible configurar la lista de reemplazo que usarán las primitivas, como muestran las líneas 2-4 y 7-8 del Listado 5.2 para los operadores AOR y ROR.

```

1 // Configuración de la lista de reemplazo para AOR
2 BIN (*) -> {/, %};
3 BIN (/) -> {*, %};
4 BIN (%) -> {*, /};
5
6 // Configuración de la lista de reemplazo para ROR
7 BIN (>) -> {<, <=, !=, ==, >=};
8 BIN (==) -> {<, <=, !=, >, >=};
9
10 // Llamadas a los operadores de mutación
11 AOR ;
12 ROR ;

```

Listado 5.2 Programa Major-Mml.

```

1 AOR "Arihtmetic operator replacement" {
2   modify one InfixExpression
3   where {operator=['*', '/', '%']}
4   with {operator=['*', '/', '%']}
5 }
6
7
8 ROR "Relational operator replacement" {
9   modify one InfixExpression
10  where {operator=['>', '==']}
11  with {operator=['<', '<=','!=','==','>','>=']}
12 }

```

Listado 5.3 Programa WODEL.

En contraste, como se ha mostrado en el Capítulo 3, `WODEL` es un DSL que permite la creación de operadores de mutación arbitrarios, y no sólo configurar listas de reemplazo de operadores. Por ejemplo, el operador de mutación presentado en la Figura 5.2 no puede definirse con `Major-Mml`, ya que requiere poder identificar llamadas a constructores. Esto demuestra la expresividad de `WODEL` en comparación con `Major-Mml`.

5.3.1.4. Eficiencia y eficacia

En esta sección, se evalúa la eficiencia (*¿cuantos mutantes por segundo puede producir `WODEL-TEST`?*) y la eficacia (*¿los mutantes generados son adecuados para evaluar los conjuntos de pruebas?*). Con este objetivo, se ha realizado un experimento consistente en la ejecución de un proceso de pruebas de mutación sobre una librería Java creada por terceros. El objetivo es evaluar hasta qué punto `WODEL-TEST` es capaz de producir resultados similares a otras herramientas de pruebas de mutación en cuanto a la eficiencia y eficacia.

El experimento ha utilizado el proyecto `functional-matrix-operator`⁶, que tiene 74 clases Java y 2 586 LOC, y un conjunto de pruebas de 10 clases Java y 647 LOC. El experimento se ejecutó en un PC con un procesador Intel Core i7-2600 y 12 GB de RAM, bajo un sistema operativo Linux Ubuntu 18.04.1 LTS. En el caso de `WODEL-TEST` para Java, el proceso de pruebas de mutación se ejecutó tres veces y se seleccionó el tiempo de la ejecución más lenta.

La Tabla 5.2 presenta un resumen de los resultados. Las columnas muestran el número de mutantes generados distinguiendo entre muertos y vivos, el mutation score, el tiempo total de ejecución, y el tiempo medio por mutante.

Herramienta	Mutantes (muertos/vivos)	Mutation score	Tiempo ejec.	Tiempo por mutante
Major	1 638 (331/864)	20.21 %	11h21min40seg	24.99seg
PITest	918 (321/597)	34.97 %	56min9seg	3.67seg
LittleDarwin	439 (130/309)	29.61 %	2h45min27seg	22.61seg
WODEL-TEST/Java	4 756 (985/3 771)	20.71 %	2h40min27seg	2.02seg

Tabla 5.2 Aplicación de las herramientas de pruebas de mutación para Java sobre el proyecto `functional-matrix-operator`.

En cuanto a la eficacia, las herramientas de pruebas de mutación obtuvieron mutation scores en un rango del 20.21 % al 34.97 %, siendo `Major` y `WODEL-TEST` los que tuvieron los valores más bajos. Hay que indicar que `WODEL-TEST` generó más mutantes, ya que sus operadores de reemplazo son más exhaustivos. Mientras que `LittleDarwin` y `PITest` sólo aplican uno de los cambios posibles considerados para el reemplazo de operadores, `WODEL-TEST` aplica todos ellos. Por ejemplo, en el caso de `AOR`, `WODEL-TEST` reemplaza cada

⁶<https://github.com/soursop/functional-matrix-operator>

aparición de los operadores aritméticos por los otros cuatro, mientras que LittleDarwin y PITest sólo efectúan un reemplazo. Otro motivo es que el conjunto de operadores de mutación de `WODEL-TEST` es el mayor dado que incluye todos los operadores definidos por el resto de herramientas. No obstante, hay que indicar que la herramienta de pruebas de mutación generada con `WODEL-TEST` permite seleccionar los operadores a aplicar (ver Figura 5.4).

La tabla no incluye los resultados de Javalanche porque, a pesar de realizar el mayor de nuestros esfuerzos, no fuimos capaces de completar una ejecución con todas las pruebas. Para obtener una ejecución de Javalanche sin errores, tuvimos que excluir algunas pruebas. En ese caso, la herramienta generó 758 mutantes y mató 75 de ellos, con un mutation score de alrededor del 10%.

Con respecto a la eficiencia, `WODEL-TEST` tardó aproximadamente 2 horas y media en ejecutar todo el proceso de pruebas de mutación, lo que supone un tiempo menor que el invertido por Major y LittleDarwin, aunque superior al de PITest. Hay que destacar que `WODEL-TEST` generó 10 veces más mutantes que LittleDarwin, 5 veces más mutantes que PITest, y 3 veces más mutantes que Major. Si analizamos el tiempo por mutante, `WODEL-TEST` fue el más rápido.

En conjunto, observando los resultados del experimento, podemos concluir que la herramienta de pruebas de mutación para Java creada con `WODEL-TEST` es comparable a las herramientas de pruebas de mutación existentes, en cuanto a la eficacia y eficiencia del proceso de pruebas de mutación.

5.3.1.5. Discusión de los resultados y aspectos que pueden afectar a la validez

La Sección 5.3.1.2 muestra que `WODEL-TEST` proporciona una funcionalidad comparable a la de las herramientas de pruebas de mutación existentes para Java, y además, ofrece un amplio conjunto de vistas interactivas con los resultados del proceso de pruebas de mutación, proporciona el código mutado para su inspección, permite la configuración de la tecnología de pruebas de unidad, e incluye mecanismos para especificar criterios de equivalencia de los mutantes. En concreto, `WODEL-TEST` puede ser una mejor opción para aplicar pruebas de mutación a programas Java que el resto de herramientas analizadas en tres situaciones: cuando el ingeniero de pruebas necesita tener acceso al código fuente de los mutantes, cuando quiere razonar sobre qué mutantes reducen el mutation score y por qué, o cuando quiere experimentar con nuevos operadores de mutación.

Una de las características destacadas de `WODEL-TEST` es su extensibilidad, ya que el DSL `WODEL` permite definir nuevos operadores de mutación. Aunque Major también se puede extender utilizando un DSL, éste es menos expresivo que `WODEL` debido a dos factores. Por una parte, Major trabaja a nivel del bytecode mientras que `WODEL` trabaja a nivel del modelo.

Por otra parte, Offutt et al. mostraron que un conjunto reducido de operadores de mutación es suficiente para producir conjuntos de pruebas de calidad semejante [95], y este conjunto no ha cambiado mucho en trabajos subsiguientes [98]. Sin embargo, las nuevas características añadidas a los lenguajes de programación existentes (p. ej., funciones lambda), y los nuevos lenguajes que han surgido recientemente, requieren la capacidad de poder de diseñar nuevos operadores de mutación.

Como muestra la Tabla 5.2, el proceso de pruebas de mutación es tan efectivo como el proceso de pruebas de mutación realizado por las otras herramientas (con mutation scores comparables, como en el caso de Major). Además, el tiempo de ejecución por mutante es similar al de PITest, que funciona a nivel de bytecode, y mucho menor que el invertido por Major y LittleDarwin.

En conjunto, podemos responder a la pregunta RQ1 de forma positiva: `WODEL-TEST` puede generar herramientas de pruebas de mutación comparables a las herramientas existentes. Las herramientas generadas tienen ventajas en términos de extensibilidad y configuración, ofreciendo un amplio conjunto de vistas interactivas con información del proceso de pruebas de mutación.

Aspectos que pueden afectar a la validez. Hay que mencionar dos aspectos que pueden afectar a la validez de los resultados obtenidos en esta evaluación. En primer lugar, se han analizado cuatro herramientas de pruebas de mutación para Java, pero puede haber otras con una funcionalidad más sofisticada o mejor eficiencia que no se han tenido en cuenta. Para mitigar este riesgo, las herramientas utilizadas en la evaluación son ampliamente conocidas en la comunidad de pruebas de mutación, y representativas de la funcionalidad y eficiencia típicas en procesos de pruebas de mutación. En segundo lugar, esta evaluación sólo considera un proyecto. En el futuro, se planea replicar el experimento con otros proyectos para incrementar la confianza en los resultados.

5.3.2. Caso de estudio: Construyendo una herramienta de pruebas de mutación para ATL

Esta sección presenta un caso de estudio sobre el desarrollo de una herramienta de pruebas de mutación para el *Atlas transformation language* (ATL) [65]. ATL es un lenguaje de transformación de modelos de uso extendido que se utiliza para definir transformaciones modelo-a-modelo. Este tipo de transformaciones traduce un modelo conforme a un meta-modelo de origen en otro modelo conforme a un meta-modelo de destino (ver Figura 5.9).

Utilizamos ATL como caso de estudio porque incluso siendo un lenguaje muy popular en MDE, hay pocas herramientas públicas para aplicar pruebas de mutación a programas ATL.

Una posible razón es el coste y la complejidad de desarrollo de este tipo de herramientas, ya que los operadores de mutación no sólo necesitan considerar programas ATL sino que también requieren realizar consultas a los meta-modelos de origen y destino. El objetivo de esta sección es mostrar cómo *WODEL-TEST* facilita la creación de una herramienta de pruebas de mutación para ATL, lo que resulta de ayuda para responder a la pregunta de investigación RQ2.

A continuación, la Sección 5.3.2.1 introduce los fundamentos de ATL. La Sección 5.3.2.2 describe cómo crear la herramienta de pruebas de mutación para ATL con *WODEL-TEST*. La Sección 5.3.2.3 evalúa el proceso de creación de la herramienta de pruebas de mutación. Por último, la Sección 5.3.2.4 comenta los resultados e identifica aspectos que pueden afectar a la validez.

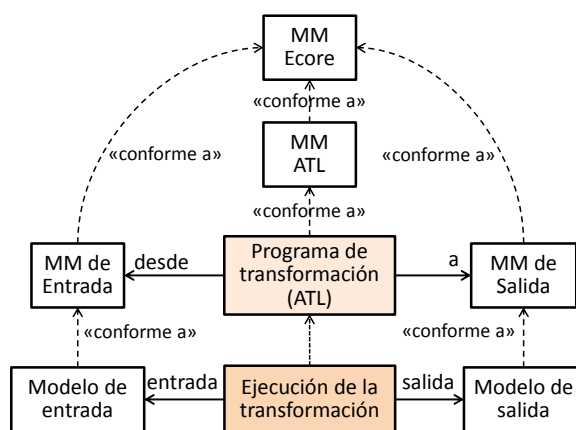


Figura 5.9 Esquema de transformaciones modelo-a-modelo con ATL.

5.3.2.1. El lenguaje ATL

ATL permite definir programas de transformación utilizando una notación textual basada en reglas. Los programas ATL son conformes al meta-modelo de ATL, y están también tipados con respecto a los meta-modelos de origen y destino de la transformación. Como muestra la Figura 5.9, los meta-modelos son conformes a Ecore, que es un lenguaje de meta-modelado que sigue el estándar Meta-Object Facility de la OMG [87].

La Figura 5.10 muestra el efecto de ejecutar un programa ATL que transforma un modelo *Families* en un modelo *Persons*. El programa simplemente crea un objeto *Male* por cada objeto *Member* que tiene el papel de *father* o de *sons* en un objeto *Family*; y un objeto *Female* por cada objeto *Member* que tiene el papel de *mother* o de *daughters*. El atributo *fullName* de los objetos creados es la concatenación de los atributos *firstName* y *lastName* de los objetos originales.

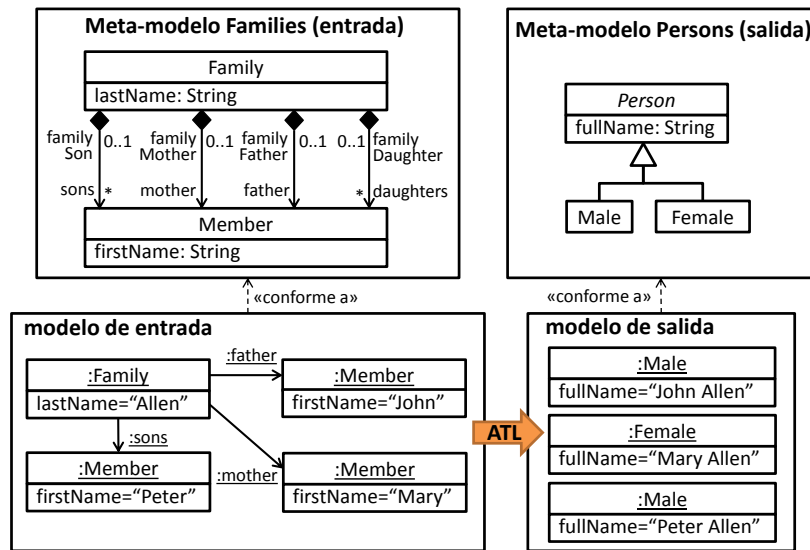


Figura 5.10 Ejemplo de transformación ATL.

Los programas ATL consisten en reglas declarativas que indican cómo un patrón de objetos en el modelo origen se traduce en un patrón de objetos en el modelo destino. Como ejemplo, el Listado 5.4 muestra un programa ATL que realiza la transformación presentada en la Figura 5.10.

Las líneas 2-4 del listado indican los meta-modelos de entrada y salida de la transformación. El programa declara dos reglas en las líneas 23-28 (`Member2Male`) y 31-36 (`Member2Female`), y dos *helpers* en las líneas 7-11 (`isFemale`) y 14-20 (`familyName`). La regla `Member2Male` se ejecuta por cada objeto masculino `Member`. Las reglas se componen de un patrón de entrada con los objetos a buscar (línea 24), un patrón de salida con los objetos a crear (línea 25), y los inicializadores de sus atributos que se denominan *bindings* (línea 26). La regla `Member2Female` es similar, pero en su caso se aplica sobre objetos `Member` femeninos y crea objetos `Female`.

Un *helper* de ATL se puede describir como el equivalente en el lenguaje ATL a un método. Mediante dichos *helpers* se puede definir fragmentos de código ATL que pueden utilizarse en distintos puntos de la transformación. Por ejemplo, el *helper* `isFemale` (líneas 7-11 en el Listado 5.4) determina si el objeto `Member` que recibe es femenino o no. Para ello, comprueba que al menos una de sus referencias `familyMother` y `familyDaughter` tiene un valor. En este caso, este *helper* devuelve `true`, y el objeto `member` recibido se ha catalogado como femenino. En caso contrario, este *helper* devuelve `false`. El *helper* `isFemale` se utiliza en las líneas 24 y 32 de la transformación presentada en el Listado 5.4.

```

1 module Families2Persons;
2 --@path Families=/Families2Persons/Families.ecore
3 --@path Persons=/Families2Persons/Persons.ecore
4 create OUT:Persons from IN:Families;
5
6 -- Devuelve true si el objeto Member es femenino
7 helper context Families!Member def: isFemale(): Boolean =
8     if not self.familyMother.ocllsUndefined() then true
9     else if not self.familyDaughter.ocllsUndefined() then true
10        else false endif
11    endif;
12
13 -- Obtiene el apellido de un objeto Member
14 helper context Families!Member def: familyName: String =
15     if not self.familyFather.ocllsUndefined() then self.familyFather.lastName
16     else if not self.familyMother.ocllsUndefined() then self.familyMother.lastName
17     else if not self.familySon.ocllsUndefined() then self.familySon.lastName
18        else self.familyDaughter.lastName endif
19    endif;
20
21 -- Crea un objeto Male a partir de un objeto Member que es masculino
22 rule Member2Male {
23     from s:Families!Member (not s.isFemale())
24     to t:Persons!Male (
25         fullName <- s.firstName + ' ' + s.familyName
26     )
27 }
28
29 -- Crea un objeto Female a partir de un objeto Member que es femenino
30 rule Member2Female {
31     from s:Families!Member (s.isFemale())
32     to t:Persons!Female (
33         fullName <- s.firstName + ' ' + s.familyName
34     )
35 }
36 }

```

Listado 5.4 Ejemplo de programa ATL que transforma modelos Families en modelos Persons.

5.3.2.2. Utilizando WODEL-TEST para construir una herramienta de pruebas de mutación para ATL

La implementación de ATL está basada en EMF. Su sintaxis abstracta se define por un meta-modelo, del que la Figura 5.11 presenta un fragmento. Este fragmento muestra que el patrón de entrada de las reglas (clase InPattern) declara una o más VariableDeclarations que representan variables con un nombre y un tipo. Este tipo debe ser un OclModelElement definido en un OclModel, o en otras palabras, una clase perteneciente al meta-modelo de entrada.

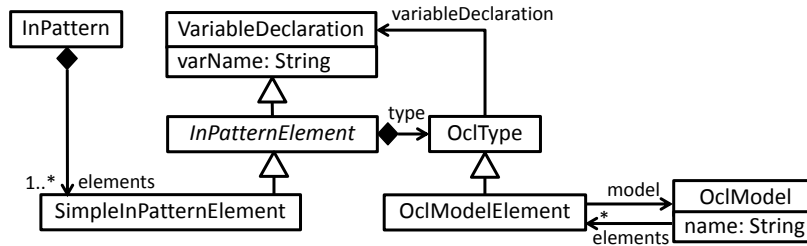


Figura 5.11 Extracto del meta-modelo de ATL.

La distribución de ATL proporciona una transformación texto-a-modelo para convertir programas ATL textuales en instancias del meta-modelo de ATL, y una transformación modelo-a-texto para generar código ATL a partir de un modelo ATL en memoria. Así, el soporte del lenguaje requerido por *WODEL-TEST* simplemente reutiliza estas dos transformaciones.

Con respecto al soporte de ejecución, *WODEL-TEST* requiere definir cómo compilar los programas ATL y cómo ejecutar los casos de prueba. Los programas ATL pueden compilarse programáticamente en bytecode por la máquina virtual de ATL. Dado que no hay un entorno estándar de pruebas unitarias para ATL, utilizamos los modelos de entrada de las transformaciones como casos de prueba. Utilizando este enfoque, no hay necesidad de especificar un oráculo explícito de pruebas. En su lugar, para cada modelo de entrada, ejecutamos las transformaciones originales y de las transformaciones mutantes y comparamos los resultados: si las salidas son diferentes, la prueba falló, en otro caso, la prueba fue superada.

El último componente de una especificación para la creación de una herramienta de pruebas de mutación se refiere al soporte de mutación. Se ha utilizado *WODEL* para definir los 18 operadores de mutación propuestos en [126]. Estos operadores, que se listan en la Tabla 5.3, permiten la creación, la eliminación, y la modificación de las reglas, elementos de patrones de entrada, elementos de patrones de salida, filtros de reglas y *bindings*. Dado que los operadores se especificaron originalmente como transformaciones ATL [126], la tabla también compara las LOC de sus codificaciones utilizando ATL y *WODEL* (en el primer caso, sólo cuando estaban disponibles en [126]).

Como ejemplo, el Listado 5.5 muestra la codificación *WODEL* del operador de mutación que crea un elemento de patrón de entrada (el Anexo H contiene la implementación de todos los demás operadores). El operador requiere la creación de una nueva declaración de variable tipada por una clase existente en el meta-modelo de entrada. Con este objetivo, la línea 3 declara un recurso adicional de nombre *input* que indica la ubicación del meta-modelo de entrada de la transformación, y es conforme al meta-modelo *Ecore*. A continuación, el

Concepto	Operador de mutación	LOC en Wodel	LOC en ATL
Regla	Creación	1	-
	Borrado	1	-
	Cambio de nombre	1	-
Elemento de patrón de entrada	Creación	6	14
	Borrado	1	-
	Cambio de tipo	4	-
	Cambio de nombre	1	-
Filtro de regla	Creación	10	-
	Borrado	1	-
	Cambio de condición	4	-
Elemento de patrón de salida	Creación	6	-
	Borrado	1	6
	Cambio de tipo	4	-
	Cambio de nombre	1	-
<i>Binding</i>	Creación	6	-
	Borrado	1	3
	Cambio de valor	2	-
	Cambio de propiedad	6	-

Tabla 5.3 Operadores de mutación para ATL.

programa selecciona una clase del meta-modelo de entrada (línea 7), y crea un elemento del patrón de entrada (línea 10) cuyo tipo es la clase seleccionada (línea 11).

```

1 generate 2 mutants in "data/out/" from "data/model/"
2 metamodel "/data/model/ATL.ecore"
3 with resources from {input="data/model/in" metamodel="/data/model/Ecore.ecore"}
4
5 with blocks {
6   cipe "Create in pattern element" {
7     cl = select one EClass from input resources
8     p = select one InPattern
9     mod = select one OclModel in p->elements->type->model
10    ipe = create SimpleInPatternElement in p->elements with {varName = random-string(2, 4)}
11    elem = create OclModelElement in ipe->type with {name = cl.name, variableDeclaration = ipe}
12    modify mod with {elements += elem}
13  }
14 }

```

Listado 5.5 Programa WODEL que implementa el operador de *creación de elemento de patrón de entrada*.

WODEL asegura que todos los mutantes generados son conformes al meta-modelo de ATL. Además, proporciona un punto de extensión para añadir otros criterios que todos los mutantes deben cumplir. En este caso, se ha implementado este punto de extensión para realizar un análisis estático del mutante, y descartarlo si tiene errores de tipado que puedan producir errores en tiempo de ejecución. Para realizar el análisis, se utiliza programáticamente un analizador estático para ATL llamado anATLyzer [105]. Esto asegura que los mutantes no tendrán errores en tiempo de ejecución.

Finalmente, se han definido criterios de equivalencia de dominio específico para programas ATL basados en la comparación del bytecode de su compilación, que se serializa a XML.

La Figura 5.12 muestra la herramienta de pruebas de mutación generada para ATL, que tiene una estructura similar a la desarrollada para Java. El editor ATL, con etiqueta 1, contiene el código de una transformación mutante. En particular, la concatenación de Strings en la línea 34 del Listado 5.4 ha sido reemplazada por un literal en la línea 62 de la Figura 5.12. En la etiqueta 2 se presenta el explorador de proyectos de Eclipse. En esta vista se muestra el proyecto ATL bajo prueba con una carpeta para cada uno de los mutantes generados (proyecto “Families2Persons”), y el conjunto de pruebas, con los dos modelos de entrada de la transformación, que se utiliza en el proceso de pruebas de mutación (proyecto “ATLCheck”). En la etiqueta 3 se muestra la vista de los resultados del proceso de mutación clasificados por mutante. Por último, en la etiqueta 4 se presenta la vista global de resultados del proceso de mutación, con los operadores de mutación aplicados y los no aplicados, los mutantes muertos/equivalentes/vivos, las pruebas superadas/no superadas, el mutation score, el número de mutantes generados y detalles del tiempo de ejecución.

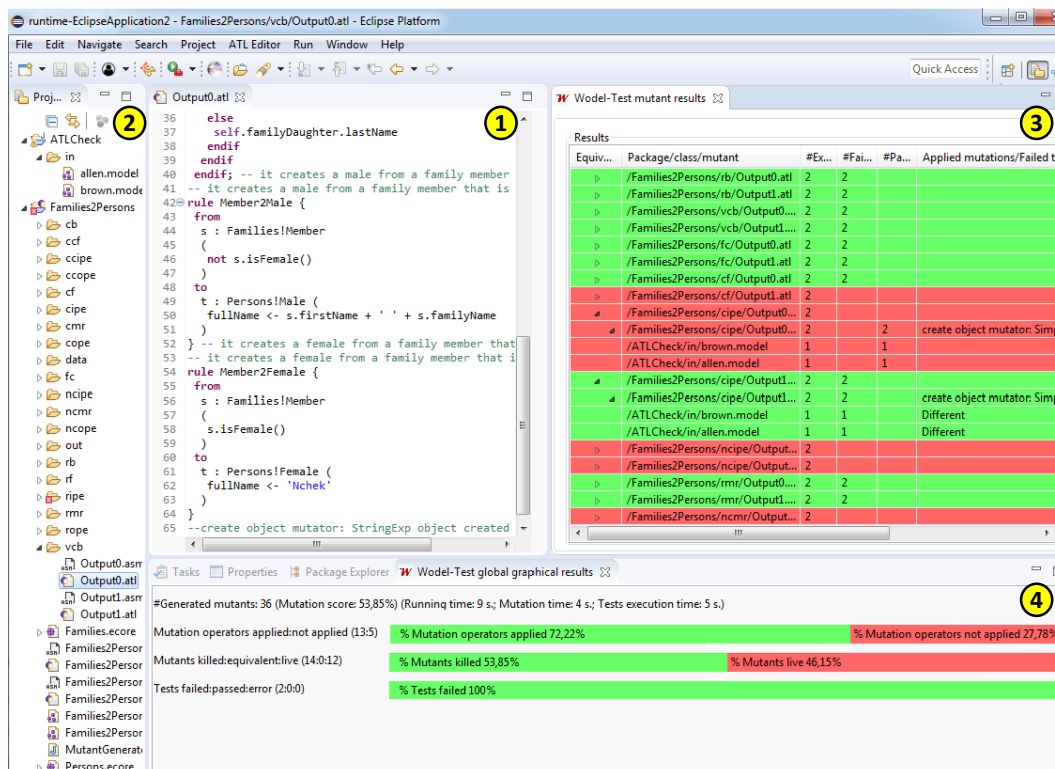


Figura 5.12 Entorno de pruebas de mutación para ATL creado con WODEL-TEST.

5.3.2.3. Evaluando el proceso de creación de la herramienta de pruebas de mutación para ATL

A continuación, se evalúa la utilización de `WODEL-TEST` para construir una herramienta de pruebas de mutación para ATL, en comparación con la técnica propuesta en [126] en la que está basado nuestro conjunto de operadores de mutación. Analizamos tres aspectos: (i) la definición de operadores de mutación, (ii) el control de la ejecución de los operadores de mutación, y (iii) la especificación y provisión de otras funcionalidades útiles para pruebas de mutación.

Definición de los operadores de mutación. Dado que los programas ATL están representados internamente como modelos, una alternativa a `WODEL` para definir operadores de mutación es la utilización de un lenguaje de transformación de modelos de propósito general. Como se ha mencionado en la sección anterior, en [126] los autores proponen 18 operadores de mutación para ATL, e implementan algunos de ellos utilizando ATL (específicamente, utilizando transformaciones de ATL de orden superior).

El Listado 5.6 muestra el programa ATL que implementa el operador de mutación de *creación de un elemento de patrón de entrada*. El programa crea un elemento de patrón de entrada con un tipo inexistente llamado “Complete_IPE” (línea 22). A continuación, otro programa ATL que se muestra en el Listado 5.7 se ejecuta en modo de refinado⁷ para modificar la transformación del mutante generada por el primer programa. Este segundo programa selecciona una clase aleatoria del meta-modelo de entrada y reemplaza el literal “Complete_IPE” por el nombre de la clase. En cambio, el código `WODEL` del mismo operador de mutación es capaz de leer el nombre de la clase del meta-modelo de entrada, es decir, no hay necesidad de definir dos programas diferentes con este objetivo (ver Listado 5.5). La forma de asignar un nombre al nuevo elemento del patrón también es diferente, dado que el programa ATL depende de un conjunto de nombres construido previamente (línea 6 en el Listado 5.6), mientras que el programa `WODEL` genera una cadena aleatoria (línea 10 del Listado 5.5).

Control de la ejecución de la mutación. El operador de mutación para ATL que muestra el Listado 5.6 codifica la ubicación en la que el operador debe aplicarse, en este caso, en la primera regla del programa ATL semilla (línea 9). En su lugar, `WODEL` proporciona un estilo más declarativo para controlar el proceso de ejecución de la mutación, permitiendo configurar el número mínimo y máximo de aplicaciones del operador, o ejecutarlo de forma exhaustiva en cada una de las posibles ubicaciones.

⁷En el modo de refinado, el modelo de entrada de la transformación se cambia in-place y se produce como salida.


```

1 module AddInPatternElement _ FirstRule;
2
3 create OUT : ATL refining IN : ATL;
4
5 — Sequence for giving new variable names to new pattern elements that are created
6 helper def : varNames : Sequence(String) = Sequence{'a','aa','b','bb','c','cc','d','dd','e','ee','f','ff','...'};
7
8 rule AddInPatternElement {
9   from s : ATL!InPattern (ATL!Rule.allInstances()→first() = s."rule") — Add SIPE only in first rule
10  to t : ATL!InPattern(
11    elements <- s.elements →append(newIPE)
12  ),
13  newIPE : ATL!SimpleInPatternElement (
14    — We have to give a variable name that no PatternElement has
15    varName <- thisModule.varNames→any(n |
16      ATL!PatternElement.allInstances()→collect(pe|pe.varName)→excludes(n)),
17    type <- newOCLType
18  ),
19  — The type is composed of a model and a name: model!name
20  newOCLType : ATL!OclModelElement(
21    model <- s.elements→first().type.model,
22    name <- 'Complete_IPE'
23  )
24 }

```

Listado 5.6 Un programa ATL que implementa el operador de *creación de un elemento de patrón de entrada* [126].

La herramienta creada con WODEL-TEST permite seleccionar los operadores de mutación a aplicar en el proceso de pruebas de mutación (ver Figura 5.4). Este control de grano fino sería difícil de emular utilizando sólo ATL debido a su semántica de ejecución. Esto es así ya que las reglas ATL se aplican exactamente una vez en cada una de las posibles coincidencias, y no se puede aplicar varias reglas sobre el mismo objeto de entrada porque esto generaría un error en tiempo de ejecución. Como consecuencia, los operadores de mutación especificados con ATL no pueden agruparse en el mismo programa de transformación, se requiere separar cada operador en varios programas orquestados manualmente o por medio de un programa controlador Java dedicado.

Por último, respecto a la concisión de las especificaciones del proceso de mutación, la herramienta prototipo proporcionada por [126] consta de alrededor de 520 LOC, mientras que la implementación del soporte LanguageServiceProvider para ATL en WODEL-TEST tiene 327 LOC.

Funcionalidad de la herramienta de pruebas de mutación. La Tabla 5.4 compara la funcionalidad de pruebas de mutación proporcionada por ambas propuestas. La herramienta de pruebas de mutación generada por WODEL-TEST proporciona las típicas funcionalidades requeridas para un entorno de pruebas de mutación completo *out-of-the-box*, incluyendo el cálculo del mutation score, la cobertura de los operadores de mutación, y otras métricas útiles. La herramienta prototipo en [126] permite la generación de mutantes de programas ATL pero

```

1 module SecondOrderHOT;
2
3 create OUT : ATL refining IN : ATL, IN_MM : IN_MM, OUT_MM : OUT_MM;
4
5 helper def : random() : Real = "#native!"java:util::Random".newInstance().nextDouble();
6
7 -- A StringExp is one of the types that can conform to the value part of a Binding.
8 -- Since the generic mutation transformation adds 'Complete_IPE' in the value part,
9 -- a StringExp is created, whose stringSymbol is 'Complete_IPE'
10 rule CompleteInMMNames {
11   from s : ATL!StringExp (s.stringSymbol = 'Complete_IPE')
12   using {
13     classes : Sequence(IN_MM!EClass) =
14       IN_MM!EClass.allInstancesFrom('IN_MM')→select(c | not c.abstract);
15   }
16   to t : ATL!StringExp(
17     -- The idea is to have in the following a random class from the input model
18     stringSymbol <- classes→at((thisModule.random()*classes→size()).floor()+1).name
19   )
20   do {
21     classes→at((thisModule.random()*classes→size()).floor()+1).name;
22   }
23 }

```

Listado 5.7 Transformación de refinado para reemplazar “Complete_IPE” por un nombre de clase del meta-modelo de entrada [126].

no proporciona mayor soporte para pruebas de mutación. Por tanto, no hay un mecanismo para detectar los mutantes equivalentes, realizar el proceso de pruebas de mutación (es decir, la ejecución de los mutantes contra los casos de prueba) e informar de los resultados (ni siquiera proporciona el mutation score). Por tanto, para obtener una herramienta de pruebas de mutación, estas funcionalidades deberían programarse a mano, algo innecesario con una herramienta como WODEL-TEST.

Con respecto al esfuerzo de desarrollo, la especificación WODEL-TEST de la herramienta de pruebas de mutación para ATL consta de 327 LOC, lo que es comparable con el esfuerzo dedicado para especificar la herramienta de pruebas de mutación para Java, que tiene 305 LOC. Es difícil hacer una comparación con las LOC del enfoque de Troya et al. [126], dado que su herramienta no proporciona soporte para el proceso de pruebas de mutación. No obstante, en comparación, si nos fijamos en las LOC de las herramientas de pruebas de mutación para Java en la Tabla 5.1, se puede observar que requieren dos órdenes de magnitud más de código que la especificación WODEL-TEST.

5.3.2.4. Discusión de los resultados y aspectos que pueden afectar a la validez

Aunque ATL se construyó utilizando técnicas MDE, implementar sus operadores de mutación utilizando un DSL tiene ventajas con respecto a utilizar un lenguaje de transformación de propósito general. Por ejemplo, WODEL tiene facilidades para clonado de objetos

	Troya et al. [126]	WODEL-TEST/ATL
Última actualización	2015	2020
Entrada para el proceso de pruebas de mutación		
UI	Línea de comandos	Plugin de Eclipse
Ámbito	Programa ATL	Programa ATL
Conjunto de pruebas	Ninguno	Modelo de entrada (configurable)
Proceso de generación de los mutantes		
N. de operadores implementados	3/18	18/18
N. de mutantes generados por operador	1	Todos los posibles
Extensibilidad de op.	No	Sí (DSL)
Artefacto mutado	Modelo	Modelo
Código del mutante	Sí	Sí
Filtrado de los mutantes	No	No
Detección de mutantes equivalentes	No	Sí (TCE)
Informes		
Tipo de informe	Ninguno	Vistas interactivas
N. de mutantes	✗	✓
Mutation score	✗	✓
Mutantes muertos/vivos	✗	Número, lista
Cobertura de operadores de mutación	✗	Número, lista
Mutantes por programa ATL	✗	✓
Pruebas por mutante	✗	✓
Mutantes por prueba	✗	✓

Tabla 5.4 Comparativa de herramientas de pruebas de mutación para ATL.

(utilizados en el operador de *creación de regla* en el Anexo H) y retipado, asignación de valores aleatorios a los atributos (utilizados en el Listado 5.5), detección automática de contenedores compatibles para objetos nuevos, comprobación del correcto cumplimiento de las restricciones OCL, y acceso a los recursos auxiliares (p. ej., el meta-modelo de entrada en el Listado 5.5).

WODEL proporciona facilidades de control de la ejecución de la mutación que podrían ser difíciles de emular utilizando un lenguaje de transformación de modelos de propósito general. Por ejemplo, WODEL permite la aplicación exhaustiva de operadores de mutación en cada una de las posibles ubicaciones, mientras que si se utiliza ATL, esto debe emularse utilizando transformaciones diferentes. Además, WODEL-TEST permite especificar criterios de equivalencia de los mutantes (p. ej., comparación basada en XML del “bytecode” de ATL).

Otra ventaja de WODEL-TEST es que las herramientas de pruebas de mutación generadas proporcionan funcionalidad avanzada, que en otro caso necesitan programarse manualmente. Como se ha mencionado previamente, en el caso de Java, las herramientas de pruebas de mutación existentes desarrolladas manualmente requieren dos órdenes de magnitud más de código que la especificación de un entorno similar utilizando WODEL-TEST. Del mismo modo, si se necesita cambiar la funcionalidad de una herramienta de pruebas de mutación creada con WODEL-TEST (p. ej., añadir operadores de mutación que consideran nuevas características

para un lenguaje objetivo debido a nuevas versiones del lenguaje, o modificar el criterio existente de equivalencia de los mutantes), estos cambios se pueden realizar en el nivel de la especificación de la herramienta de pruebas de mutación, de modo que la herramienta de pruebas de mutación puede regenerarse a partir de esta especificación. Esto simplifica el mantenimiento de las herramientas de pruebas de mutación creadas con WODEL-TEST, dado que su especificación es de más alto nivel y por tanto más sencilla de modificar que el código equivalente.

Por último, las herramientas de pruebas de mutación creadas con WODEL-TEST son plugins de Eclipse, y por tanto, pueden integrarse de forma sencilla con otras herramientas a través de puntos de extensión (un mecanismo no intrusivo proporcionado por Eclipse que permite añadir nuevas funcionalidades a plugins existentes). En particular, WODEL-TEST define puntos de extensión para permitir a los desarrolladores integrar otras herramientas en casi cada uno de los pasos del proceso de pruebas de mutación: validación de la corrección de los mutantes, detección de los mutantes equivalentes, compilación de los mutantes, etc. Como ejemplo, la herramienta de pruebas de mutación desarrollada para ATL integra un analizador estático para comprobar la corrección de los mutantes generados y así poder descartarlos si pueden provocar errores de ejecución.

En conjunto, basándonos en las LOC necesarias para construir las herramientas de pruebas de mutación manualmente o utilizando WODEL-TEST, y basándonos en la funcionalidad que WODEL y WODEL-TEST proporcionan *out-of-the-box*, podemos responder la pregunta de investigación RQ2 afirmando que utilizar WODEL-TEST es más efectivo que construir una herramienta de pruebas de mutación desde cero utilizando un lenguaje de programación, o una combinación de lenguajes de programación y de transformación. Además, las herramientas de pruebas de mutación construidas con WODEL-TEST también tienen ventajas en términos de mantenimiento e integrabilidad con herramientas externas.

Aspectos que pueden afectar a la validez. Para definir las herramientas de pruebas de mutación para Java y ATL utilizando WODEL-TEST, se han reutilizado meta-modelos y transformaciones modelo-a-texto y texto-a-modelo existentes. Si estos artefactos no estuviesen disponibles, construir las herramientas de pruebas de mutación podría haber requerido más esfuerzo. No obstante, como se mencionaba en la Sección 5.1, el incremento de la utilización de MDE para sistemas de reingeniería existentes aumenta las posibilidades de encontrar estos artefactos para lenguajes de programación, mientras que siempre suelen estar disponibles para lenguajes de modelado.

Capítulo 6

Conclusiones y Trabajo Futuro

Esta tesis propone contribuciones en las áreas de las técnicas de mutación, la generación automática de ejercicios y la generación de herramientas de pruebas de mutación independientes del lenguaje. En el resto de este capítulo, la Sección 6.1 presenta las conclusiones de esta tesis que resumen las soluciones y los resultados de cada una de las contribuciones de este trabajo. A continuación, la Sección 6.2 analiza y presenta algunas líneas de trabajo futuro para mejorar estas contribuciones.

6.1. Conclusiones

La principal contribución de esta tesis es el DSL `WODEL` y la herramienta para desarrollar con este lenguaje. El objetivo de `WODEL` es facilitar la mutación de modelos, y dar soporte a la creación de aplicaciones que utilizan mutaciones en alguna parte de su proceso. `WODEL` proporciona tanto primitivas de mutación para la creación, modificación, borrado, retipado y clonado de objetos, como aquellas necesarias para la creación, modificación, y borrado de referencias. La herramienta `WODEL` consiste en un IDE completo para desarrollar programas `WODEL`, con un asistente para completado de código, resaltado de sintaxis y compilación automática de los programas a código Java. La herramienta desarrollada proporciona también funcionalidades útiles para la mutación de modelos, como la validación de los mutantes generados, un registro de las mutaciones aplicadas, la detección de mutantes equivalentes, y un punto de extensión para utilizar los mutantes en aplicaciones de post-procesado. Además, la herramienta `WODEL` proporciona funcionalidades para una ingeniería sistemática de los operadores de mutación, como la síntesis de modelos semilla y las métricas de mutación estáticas y dinámicas.

Esta tesis incluye además dos aplicaciones implementadas como extensiones de `WODEL` que muestran el potencial de la herramienta. La primera es la herramienta `WODEL-EDU` para

generación automática de ejercicios. La solución al ejercicio es un modelo que se muta, y los ejercicios muestran la solución original y sus mutantes para que el estudiante seleccione la opción correcta. La configuración de los ejercicios se lleva a cabo utilizando DSLs. Hasta la fecha se ha aplicado a autómatas, pero el lenguaje de los ejercicios es configurable mediante un meta-modelo, y por tanto se puede aplicar a otros dominios. Los resultados obtenidos de la evaluación de WODEL-EDU muestran que de acuerdo con la apreciación de los participantes, los ejercicios generados por WODEL-EDU son útiles para el aprendizaje de la materia.

La segunda extensión de WODEL presentada en esta tesis es la herramienta WODEL-TEST para generación de herramientas de pruebas de mutación para lenguajes de programación y modelado. Se debe proporcionar un meta-modelo del lenguaje y las transformaciones entre la sintaxis concreta del lenguaje y su meta-modelo; los operadores de mutación se especifican con WODEL; y se pueden configurar los criterios para la detección de mutantes equivalentes. Como prueba de concepto, se han implementado dos herramientas de pruebas de mutación utilizando WODEL-TEST, la primera para Java, y la segunda para ATL. La herramienta de pruebas de mutación generada para Java tiene un rendimiento similar al de otras herramientas existentes, mejorando en el aspecto de la interactividad y la cantidad de información recopilada sobre el proceso de pruebas de mutación. La herramienta de pruebas de mutación generada para ATL supone una contribución novedosa, dado que en la actualidad no se ha identificado ninguna otra herramienta que dé un soporte integral a este proceso para ATL.

6.2. Trabajo futuro

Como trabajo futuro, se plantean las siguientes nuevas aportaciones en las tres líneas principales de esta tesis.

En la actualidad, estamos extendiendo el proceso de síntesis de modelos de dos formas. La primera consiste en generar modelos para los que los operadores no son aplicables pero que están cerca de ser aplicables, denominados *near misses* [91]. Este enfoque resulta útil para comprobar que los operadores de mutación sólo tienen aplicación sobre el objetivo para el que se han diseñado, es decir, al aplicarlos sobre los modelos *near misses* no deberían generar ningún mutante. En el caso de que estos operadores de mutación sí generen mutantes al aplicarse sobre estos modelos *near misses*, hay que afinarlos. La segunda consiste en generar los modelos semilla asegurando que la ejecución del programa WODEL proporcionará un modelo correcto. Con este objetivo, se podrían utilizar técnicas para anticipar las restricciones OCL como precondiciones, basándose en [33]. Otra línea de trabajo futuro que también está orientada al desarrollador de programas de mutación es la de utilizar técnicas de análisis

estático, p. ej., para detectar conflictos y dependencias entre los operadores. Un conflicto entre operadores de mutación es, p. ej., cuando queremos modificar un objeto que ha sido borrado previamente.

En primer lugar, respecto a la expresividad de `WODEL`, planeamos extender su sintaxis para permitir la reutilización de operadores de mutación genéricos entre lenguajes similares (p. ej., los operadores definidos para un meta-modelo de C++ pueden reutilizarse con un meta-modelo de Java) utilizando técnicas de reutilización como las propuestas en [25]. Esto implica extender el DSL `WODEL` para dar soporte a la definición de funciones que pueden reutilizarse para distintos meta-modelos e invocarse desde distintos operadores de mutación.

Hasta la fecha, hemos aplicado `WODEL-EDU` a la generación automática de ejercicios de autómatas, pero planeamos aplicarlo a la generación de ejercicios en otros dominios. El estudio con usuarios realizado para evaluar la calidad de los ejercicios generados muestra que, si bien los estudiantes consideran que los ejercicios son útiles para aprender autómatas, deben mejorarse algunos aspectos relacionados con la usabilidad de la interfaz de usuario de la aplicación web y la facilidad de comprensión de los ejercicios, en concreto para los ejercicios de selección de reparación múltiple. También planteamos como trabajo futuro ampliar los lenguajes del plugin `WODEL-EDU` para dar soporte a entornos de aprendizaje más complejos o para otras plataformas (ej. apps). Planteamos realizar nuevos estudios con usuarios con más participantes, en los que compararemos los resultados obtenidos en los ejercicios generados con `WODEL-EDU` con otros creados manualmente.

Respecto a `WODEL-TEST`, en la actualidad estamos trabajando en mejorar su eficiencia para abordar programas más grandes (el programa más grande que se ha considerado en la evaluación es de unas 4 000 LOC). En el futuro, nos gustaría realizar un estudio con usuarios para analizar la usabilidad de las herramientas de pruebas de mutación generadas. Otro objetivo es dotar a `WODEL-TEST` de puntos de extensión para dar soporte al filtrado de mutantes (p. ej., basados en la cobertura de instrucciones), técnicas de reducción (p. ej., basadas en muestras) y control de los mutantes redundantes (p. ej., una aproximación dinámica del mutation score disjunto [78]). Otra línea de trabajo futuro que merece la pena investigar incluye la automatización de la síntesis de las transformaciones modelo-a-texto y texto-a-modelo a partir del meta-modelo del lenguaje objetivo e instancias de ejemplo de los programas textuales.

Nuestra propuesta de trabajo futuro incluye también desarrollar post-procesadores a `WODEL` para otras áreas como la computación evolutiva o la resolución automática de problemas de lógica, que pueden además suponer mejoras en el DSL `WODEL`.

Bibliografía

- [1] *Architecture driven modernization*. <https://www.omg.org/adm/>. Last accessed: 2020-Feb.
- [2] A Graphical Tooling Infrastructure (Graphiti) website. <https://www.eclipse.org/graphiti/>. Last accessed: 2020-Feb.
- [3] Acceleo website. <https://www.eclipse.org/acceleo/>. Last accessed: 2020-Feb.
- [4] ACME platform website. <http://acme.udg.edu/es/equip.php>. Last accessed: 2020-Feb.
- [5] Agrawal, H., DeMillo, R. A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E., Martin, R., Mathur, A. P. y Spafford, E.: *Design of mutant operators for the C programming language*. Informe técnico., Purdue University, 1989.
- [6] Aichernig, B. K., Brandl, H., Jöbstl, E. y Krenn, W.: *UML in action: A two-layered interpretation for testing*. SIGSOFT Softw. Eng. Notes, 36(1):1–8, Ene. 2011, ISSN 0163-5948.
- [7] Alhwikem, F., Paige, R. F., Rose, L. y Alexander, R.: *A systematic approach for designing mutation operators for MDE languages*. En *13th Workshop on Model-Driven Engineering, Verification and Validation co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MoDELS*, págs. 54–59, 2016.
- [8] Allweyer, T.: *BPMN 2.0*. BoD, 2010, ISBN 3839149851, 9783839149850.
- [9] Alsmadi, I. M.: *Using mutation to enhance GUI testing coverage*. IEEE Software, 30(1):67–73, 2013.
- [10] Aranega, V., Mottu, J., Etien, A., Degueule, T., Baudry, B. y Dekeyser, J.: *Towards an automation of the mutation analysis dedicated to model transformation*. STVR, 25(5-7):653–683, 2015.
- [11] Arts, T., Hughes, J., Johansson, J. y Wiger, U.: *Testing telecoms software with Quviq QuickCheck*. En *ACM SIGPLAN Workshop on Erlang, ERLANG*, págs. 2–10, New York, NY, USA, 2006. ACM, ISBN 1-59593-490-1.
- [12] ATL (Atlas Transformation Language) website. <https://eclipse.org/atl/>. Last accessed: 2020-Feb.
- [13] Aycock, J.: *A Brief History of Just-in-time*. ACM Comput. Surv., 35(2):97–113, 2003, ISSN 0360-0300.

- [14] Aziz, B.: *Towards a mutation analysis of IoT protocols*. Information & Software Technology, 100:183–184, 2018.
- [15] Baker, R. y Habli, I.: *An empirical evaluation of mutation testing for improving the test quality of safety-critical software*. IEEE Trans. Software Eng., 39(6):787–805, 2013.
- [16] Bartel, A., Baudry, B., Munoz, F., Klein, J., Mouelhi, T. y Traon, Y. L.: *Model driven mutation applied to adaptative systems testing*. En *Proc. Mutation Analysis Workshop*, págs. 408–413, 2011.
- [17] Benac Earle, C., Fredlund, L. A. y Hughes, J.: *Automatic grading of programming exercises using property-based testing*. En *ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE, págs. 47–52, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4231-5.
- [18] Bettini, L.: *Implementing domain specific languages with Xtext and Xtend - Second edition*. Packt Publishing, 2nd ed., 2016, ISBN 1786464969, 9781786464965. Available at <https://www.eclipse.org/Xtext/>.
- [19] Blumenstein, M., Green, S., Nguyen, A. y Muthukkumarasamy, V.: *An experimental analysis of GAME: A generic automated marking environment*. En *9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE, págs. 67–71, New York, NY, USA, 2004. ACM, ISBN 1-58113-836-9.
- [20] Bombieri, N., Fummi, F., Guarnieri, V. y Pravadelli, G.: *Testbench qualification of SystemC TLM protocols through mutation analysis*. IEEE Transactions on Computers, 63(5):1248–1261, May 2014, ISSN 0018-9340.
- [21] Bonsangue, M. M., Kok, J. N. y Sere, K.: *An approach to object-orientation in action systems*. En Jeuring, J. (ed.): *Mathematics of Program Construction*, págs. 68–95, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg, ISBN 978-3-540-69345-1.
- [22] BPMN (Business Process Model and Notation) website. <http://www.bpmn.org/>. Last accessed: 2020-Feb.
- [23] Bradbury, J. S., Cordy, J. R. y Dingel, J.: *Mutation operators for concurrent Java (J2SE 5.0)*. En *2nd Workshop on Mutation Analysis, Mutation 2006 - ISSRE Workshops 2006*, págs. 83–92, 2006.
- [24] Brambilla, M., Cabot, J. y Wimmer, M.: *Model-Driven software engineering in practice, Second edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.
- [25] Bruel, J., Combemale, B., Guerra, E., Jézéquel, J., Kienzle, J., Lara, J. de, Mussbacher, G., Syriani, E. y Vangheluwe, H.: *Model transformation reuse across metamodels - A classification and comparison of approaches*. En *International Conference on Model Transformation, ICMT*, vol. 10888 de LNCS, págs. 92–109. Springer, 2018.
- [26] Brunelière, H., Cabot, J., Dupé, G. y Madiot, F.: *MoDisco: A model driven reverse engineering framework*. Information and Software Technology, 56(8):1012–1032, 2014.

- [27] Cañizares, P. C., Núñez, A. y Merayo, M. G.: *Mutomvo: mutation testing framework for simulated cloud and HPC environments*. Journal of Systems and Software, 143:187–207, 2018.
- [28] Ceri, S., Fraternali, P. y Bongio, A.: *Web Modeling Language (WebML): a modeling language for designing Web sites*. Computer Networks, 33(1):137 – 157, 2000, ISSN 1389-1286.
- [29] Clarisó, R. y Cabot, J.: *Fixing defects in integrity constraints via constraint mutation*. En *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, págs. 74–82, Sep. 2018.
- [30] Clark, J. A., Dan, H. y Hierons, R. M.: *Semantic mutation testing*. Science of Computer Programming, 78(4):345–363, 2013, ISSN 0167-6423. Special section on Mutation Testing and Analysis (Mutation 2010) Special section on the Programming Languages track at the 25th ACM Symposium on Applied Computing.
- [31] Coles, H., Laurent, T., Henard, C., Papadakis, M. y Ventresque, A.: *PIT: A practical mutation testing tool for Java (Demo)*. En *International Symposium on Software Testing and Analysis (ISSTA)*, págs. 449–452. ACM, 2016, ISBN 978-1-4503-4390-9.
- [32] Cordy, J. R., Dean, T. R., Malton, A. J. y Schneider, K. A.: *Source transformation in software engineering using the TXL transformation system*. Information and Software Technology, 44(13):827 – 837, 2002, ISSN 0950-5849. Special Issue on Source Code Analysis and Manipulation, SCAM.
- [33] Cuadrado, J. S., Guerra, E., Lara, J. de, Clarisó, R. y Cabot, J.: *Translating target to source constraints in model-to-model transformations*. En *MODELS*, págs. 12–22. IEEE Computer Society, 2017.
- [34] Delamaro, M. E. y Maldonado: *Proteum – A tool for the assessment of test adequacy for C programs*. En *Conf. Performability in Computing Systems*, págs. 79–95, 1996.
- [35] Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A. y Domínguez-Jiménez, J. J.: *Assessment of class mutation operators for C++ with the MuCPP mutation system*. Information & Software Technology, 81:169–184, 2017.
- [36] DeMillo, R. A., Guindi, D. S., McCracken, W. M., Offutt, A. J. y King, K. N.: *An extended overview of the Mothra software testing environment*. En *Workshop on Software Testing, Verification, and Analysis*, págs. 142–151, 1988.
- [37] DeMillo, R. A., Lipton, R. J. y Sayward, F. G.: *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, 11(4):34–41, 1978.
- [38] Deng, L., Offutt, A. J., Ammann, P. y Mirzaei, N.: *Mutation operators for testing Android apps*. Information & Software Technology, 81:154–168, 2017.
- [39] Devroey, X., Perrouin, G., Schobbens, P. y Heymans, P.: *Poster: VIBeS, transition system mutation made easy*. En *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, págs. 817–818, May 2015.
- [40] Eclipse Sirius website. <https://www.eclipse.org/sirius/>. Last accessed: 2020-Feb.

- [41] Ehrig, H., Ehrig, K., Prange, U. y Taentzer, G.: *Fundamentals of algebraic graph transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.
- [42] EMFText website. <https://marketplace.eclipse.org/content/emftext>. Last accessed: 2020-Feb.
- [43] Estero-Botaro, A., Palomo-Lozano, F. y Medina-Bulo, I.: *Mutation operators for WS-BPEL 2.0*. En *International Conference on Software & Systems Engineering and their Applications, ICSSEA*, 2008.
- [44] Fabbri, S. C. P. F., Maldonado, J. C., Masiero, P. C., Delamaro, M. E. y Wong, W. E.: *Mutation testing applied to validate specifications based on Petri Nets*. En *International Conference on Formal Description Techniques*, vol. 43 de *IFIP Conference Proceedings*, págs. 329–337. Chapman & Hall, 1995.
- [45] Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T. y Masiero, P. C.: *Mutation testing applied to validate specifications based on statecharts*. En *International Symposium on Software Reliability Engineering, ISSRE*, pág. 210, 1999.
- [46] Fabbri, S. P. F., Delamaro, M. E., Maldonado, J. C. y Masiero, P.: *Mutation analysis testing for finite state machines*. En *5th International Symposium on Software Reliability Engineering*, págs. 220–229, 1994.
- [47] Ferenc, R., Beszedes, A., Tarkiainen, M. y Gyimothy, T.: *Columbus - reverse engineering tool and schema for C++*. En *International Conference on Software Maintenance*, págs. 172–181, 2002.
- [48] Fleck, M., Troya, J. y Wimmer, M.: *Search-based model transformations with MOMoT*. En Van Gorp, P. y Engels, G. (eds.): *Theory and Practice of Model Transformations*, págs. 79–87, Cham, 2016. Springer International Publishing, ISBN 978-3-319-42064-6.
- [49] Free Problem Set (FPS). <http://code.google.com/p/freeproblemset/>. Last accessed: 2020-Feb.
- [50] Gómez-Abajo, P., Guerra, E. y Lara, J. de: *A domain-specific language for model mutation and its application to the automated generation of exercises*. *Computer Languages, Systems & Structures*, 49:152 – 173, 2017, ISSN 1477-8424.
- [51] Gómez-Abajo, P., Guerra, E., Lara, J. de y Merayo, M. G.: *A tool for domain-independent model mutation*. *Science of Computer Programming*, 163:85–92, 2018.
- [52] Gómez-Abajo, P., Guerra, E., Lara, J. de y Merayo, M. G.: *Towards a model-driven engineering solution for language independent mutation testing*. En *Jornadas de Ingeniería del Software y Bases de Datos, JISBD*, pág. 4pps. Biblioteca digital SISTEDES, 2018.
- [53] Gómez-Abajo, P., Guerra, E., Lara, J. de y Merayo, M. G.: *Mutation Testing for DSLs (Tool Demo)*. En *ACM SIGPLAN International Workshop on Domain-Specific Modeling, DSM*, págs. 60–62. ACM, 2019.

- [54] Graphical Modeling Framework (GMF) website. <https://www.eclipse.org/gmf-tooling/>. Last accessed: 2020-Feb.
- [55] Groenewegen, D. M. y Visser, E.: *Integration of data validation and user interface concerns in a DSL for web applications*. *Software and System Modeling*, 12(1):35–52, 2013.
- [56] Guerra, E., Sánchez Cuadrado, J. y de Lara, J.: *Towards effective mutation testing for ATL*. En *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems, MoDELS*, págs. 78–88, Sep. 2019.
- [57] Hamlet, R. G.: *Testing programs with the aid of a compiler*. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.
- [58] Harrand, N., Fleurey, F., Morin, B. y Husa, K. E.: *ThingML: a language and code generation framework for heterogeneous targets*. En *19th International Conference on Model Driven Engineering Languages and Systems, MoDELS*, págs. 125–135, 2016.
- [59] Henard, C., Papadakis, M. y Le Traon, Y.: *Mutation-based generation of software product line test configurations*. En Le Goues, C. y Yoo, S. (eds.): *Search-Based Software Engineering*, págs. 92–106, Cham, 2014. Springer International Publishing, ISBN 978-3-319-09940-8.
- [60] Henard, C., Papadakis, M. y Traon, Y. L.: *MutaLog: A tool for mutating logic formulas*. En *Proc. International Conference on Software Testing, Verification and Validation Workshops, ICSTW*, págs. 399–404. IEEE CS, 2014.
- [61] Hierons, R. M. y Merayo, M. G.: *Mutation testing from probabilistic and stochastic finite state machines*. *Journal of Systems and Software*, 82(11):1804–1818, 2009.
- [62] Jackson, D.: *Alloy: a language and tool for exploring software designs*. *Commun. ACM*, 62(9):66–76, 2019.
- [63] JET website. <https://www.eclipse.org/modeling/m2t/?project=jet>. Last accessed: 2020-Feb.
- [64] Jia, Y. y Harman, M.: *MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language*. En *Testing: Academic Industrial Conference - Practice and Research Techniques (TAIC part 2008)*, págs. 94–98, 2008.
- [65] Jouault, F., Allilaire, F., Bézivin, J. y Kurtev, I.: *ATL: A model transformation tool*. *Science of Computer Programming*, 72(1):31 – 39, 2008, ISSN 0167-6423. Special Issue on Second issue of experimental software and toolkits, EST.
- [66] Just, R.: *The Major mutation framework: efficient and scalable mutation analysis for Java*. En *International Symposium on Software Testing and Analysis, ISSTA*, págs. 433–436. ACM, 2014, ISBN 978-1-4503-2645-2.
- [67] Kelly, S. y Tolvanen, J. P.: *Domain-specific modeling: Enabling full code generation*. Wiley, 2008, ISBN 978-0-470-03666-2.

- [68] Khan, Y. y Hassine, J.: *Mutation operators for the Atlas Transformation Language*. En *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, págs. 43–52, March 2013.
- [69] Kim, S., Clark, J. A. y McDermid, J. A.: *Investigating the effectiveness of object-oriented testing strategies using the mutation method*. *Softw. Test., Verif. Reliab.*, 11(3):207–225, 2001.
- [70] King, K. N. y Offutt, A. J.: *A Fortran language system for mutation-based software testing*. *Softw., Pract. Exper.*, 21(7):685–718, 1991.
- [71] Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Traon, Y. L. y Harman, M.: *Detecting trivial mutant equivalences via compiler optimisations*. *IEEE Trans. Software Eng.*, 44(4):308–333, 2018.
- [72] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N. y Traon, Y. L.: *How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults*. *Empirical Software Engineering*, 23(4):2426–2463, 2018.
- [73] Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E. y Brandl, H.: *MoMut: UML Model-based mutation testing for UML*. En *IEEE International Conference on Software Testing, Verification and Validation, ICST*, págs. 1–8, 2015.
- [74] Kuhlmann, M. y Gogolla, M.: *From UML and OCL to relational logic and back*. En *MODELS*, vol. 7590 de *LNCS*, págs. 415–431. Springer, 2012.
- [75] Kumar Mandal, A., Mandal, C. y Reade, C.: *Architecture of an automatic program evaluation system*. Ene. 2006.
- [76] Kusano, M. y Wang, C.: *CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications*. En *IEEE/ACM International Conference on Automated Software Engineering, ASE*, págs. 722–725. IEEE Press, 2013, ISBN 978-1-4799-0215-6.
- [77] Lackner, H. y Schmidt, M.: *Towards the assessment of software product line tests: A mutation system for variable systems*. En *18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2, SPLC*, págs. 62–69, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2739-8.
- [78] Laurent, T., Papadakis, M., Kintis, M., Henard, C., Traon, Y. L. y Ventresque, A.: *Assessing and improving the mutation testing practice of PIT*. En *International Conference on Software Testing, Verification and Validation, ICST*, págs. 430–435, 2017.
- [79] Lee, S. C. y Offutt, A. J.: *Generating test cases for XML-based web component interactions using mutation analysis*. En *International Symposium on Software Reliability Engineering, ISSRE*, págs. 200–209, 2001.
- [80] López, S., Alfonzo, G. A., Perez, O., Gonzalez, S. y Montes, R.: *A metamodel to carry out reverse engineering of C++ code into UML sequence diagrams*. *Electronics, Robotics and Automotive Mechanics Conference, CERMA*, 2:331–336, 2006.

- [81] Ma, Y., Kwon, Y. R. y Offutt, J.: *Inter-class mutation operators for Java*. En *13th International Symposium on Software Reliability Engineering, ISSRE*, págs. 352–366, 2002.
- [82] Ma, Y. S., Offutt, A. J. y Kwon, Y. R.: *MuJava: A mutation system for Java*. En *International Conference on Software Engineering, ICSE*, págs. 827–830, 2006.
- [83] Marand, E. A., Marand, E. A. y Challenger, M.: *DSMLACP: A Domain-specific modeling language for concurrent programming*. *Computer Languages, Systems & Structures*, 44:319–341, 2015.
- [84] Mariya, F. y Barkhas, D.: *A comparative analysis of mutation testing tools for Java*. En *IEEE East-West Design Test Symposium, EWDTS*, págs. 1–3, 2016.
- [85] Mens, T. y Gorp, P. V.: *A Taxonomy of Model Transformation*. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
- [86] Mernik, M., Heering, J. y Sloane, A. M.: *When and how to develop domain-specific languages*. *ACM Comput. Surv.*, 37(4):316–344, Dic. 2005, ISSN 0360-0300.
- [87] MetaObject Facility (MOF). <http://www.omg.org/mof/>. Last accessed: 2020-Feb.
- [88] Mirshokraie, S., Mesbah, A. y Pattabiraman, K.: *Guided mutation testing for JavaScript web applications*. *IEEE Trans. Software Eng.*, 41(5):429–444, 2015.
- [89] Moawad, A., Hartmann, T., Fouquet, F., Nain, G., Klein, J. y Bourcier, J.: *Polymer: A model-driven approach for simpler, safer, and evolutive multi-objective optimization development*. En *2015 3rd International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, págs. 1–8, Feb 2015.
- [90] Molina, J. G., Rubio, F. O. G., Pelechano, V., Vallecillo, A., Vara, J. M. y Vicente-Chicote, C.: *Desarrollo de software dirigido por modelos*. Ra-Ma, España, 2013.
- [91] Montaghani, V. y Rayside, D.: *Bordeaux: A tool for thinking outside the box*. En *FASE*, vol. 10202 de *LNCS*, págs. 22–39. Springer, 2017.
- [92] Mottu, J. M., Baudry, B. y Le Traon, Y.: *Mutation analysis testing for model transformations*. En Rensink, A. y Warmer, J. (eds.): *Model Driven Architecture – Foundations and Applications*, págs. 376–390, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg, ISBN 978-3-540-35910-4.
- [93] Nguyen, P. H., Papadakis, M. y Rubab, I.: *Testing delegation policy enforcement via mutation analysis*. En *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, págs. 34–42, March 2013.
- [94] Object Management Group: *UML 2.4 OCL Specification*. <http://www.omg.org/spec/OCL/>, Last accessed: 2020-Feb.
- [95] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H. y Zapf, C.: *An experimental determination of sufficient mutant operators*. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996, ISSN 1049-331X.

- [96] Offutt, A. J., Voas, J. y Payne, J.: *Mutation operators for Ada*. Informe técnico., Information and Software Systems Engineering, George Mason University, 1996.
- [97] Parsai, A., Murgia, A. y Demeyer, S.: *LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex Java systems*. En *Fundamentals of Software Engineering, FSEN*, vol. 10522 de LNCS, págs. 148–163. Springer, 2017.
- [98] Parsai, A., Murgia, A. y Demeyer, S.: *LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex Java Systems*, págs. 148–163. Springer International Publishing, Cham, 2017.
- [99] Pietsch, P., Yazdi, H. S. y Kelter, U.: *Controlled generation of models with defined properties*. En Jähnichen, S., Küpper, A. y Albayrak, S. (eds.): *Software Engineering*, págs. 95–106, Bonn, 2012. Gesellschaft für Informatik e.V.
- [100] Prähofer, H., Schatz, R., Wirth, C., Hurnaus, D. y Mössenböck, H.: *Monaco - A domain-specific language solution for reactive process control programming with hierarchical components*. *Computer Languages, Systems & Structures*, 39(3):67–94, 2013.
- [101] Queirós, R. A. P. y Leal, J. P.: *PETCHA: A programming exercises teaching assistant*. En *17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, págs. 192–197, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1246-2.
- [102] QVT specification. <http://www.omg.org/spec/QVT/>. Last accessed: 2020-Feb.
- [103] Rose, L. M., Herrmannsdoerfer, M., Mazanek, S., Gorp, P. V., Buchwald, S., Horn, T., Kalnina, E., Koch, A., Lano, K., Schätz, B. y Wimmer, M.: *Graph and model transformation tools for model migration - Empirical results from the transformation tool contest*. *Software and System Modeling*, 13(1):323–359, 2014.
- [104] Sadigh, D., Seshia, S. A. y Gupta, M.: *Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems*. En *WESE*, págs. 2:1–2:8. ACM, 2013.
- [105] Sánchez Cuadrado, J., Guerra, E. y Lara, J. de: *Static analysis of model transformations*. *IEEE Trans. Software Eng.*, 43(9):868–897, 2017.
- [106] Saritas, H. B. y Kardas, G.: *A model driven architecture for the development of smart card software*. *Computer Languages, Systems & Structures*, 40(2):53–72, 2014.
- [107] Schmidt, D. A.: *Programming language semantics*. En *Computing Handbook, Third Edition: Computer Science and Software Engineering*, págs. 69: 1–19, 2014.
- [108] Schuler, D. y Zeller, A.: *Javalanche: Efficient mutation testing for Java*. En *Joint Meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, págs. 297–298. ACM, 2009.
- [109] Sen, S. y Baudry, B.: *Mutation-based model synthesis in model driven engineering*. En *Workshop on Mutation Analysis (Mutation)*, 2006.

- [110] Shin, D., Jee, E. y Bae, D.: *Comprehensive analysis of FBD test coverage criteria using mutants*. Software and System Modeling, 15(3):631–645, 2016.
- [111] Silberschatz, A., Korth, H. y Sudarshan, S.: *Database systems concepts*. McGraw-Hill, Inc., New York, NY, USA, 5ª ed., 2006, ISBN 0072958863, 9780072958867.
- [112] Silva, A. R. da: *Model-driven engineering: A survey supported by the unified conceptual model*. Computer Languages, Systems Structures, 43:139–155, 2015.
- [113] Simão, A. d. S. y Maldonado, J. C.: *MuDeL: a language and a system for describing and generating mutants*. Journal of the Brazilian Computer Society, 8:73 – 86, Jul. 2002, ISSN 0104-6500.
- [114] Simulink website. <https://www.mathworks.com/products/simulink.html>. Last accessed: 2020-Feb.
- [115] Soler, J., Boada, I., Prados, F., Poch, J. y Fabregat, R.: *A web-based e-learning tool for UML class diagrams*. En *IEEE EDUCON 2010 Conference*, págs. 973–979, April 2010.
- [116] Spoofox website. <https://www.metaborg.org/en/latest/>. Last accessed: 2020-Feb.
- [117] Srivatanakul, T., Clark, J. A., Stepney, S. y Polack, F.: *Challenging formal specifications by mutation: A CSP security example*. En *Asia-Pacific Software Engineering Conference, APSEC*, págs. 340–350, 2003.
- [118] Steinberg, D., Budinsky, F., Paternostro, M. y Merks, E.: *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd ed., 2009, ISBN 0321331885. Available at <https://www.eclipse.org/modeling/emf/>.
- [119] Stephan, M., Alafi, M. H., Stevenson, A. y Cordy, J. R.: *Using mutation analysis for a model-clone detector comparison framework*. En *2013 35th International Conference on Software Engineering, ICSE*, págs. 1261–1264, May 2013.
- [120] Stephan, M. y Cordy, J. R.: *MuMonDE: A framework for evaluating model clone detectors using model mutation analysis*. Software Testing, Verification and Reliability, 29(1-2):e1669, 2019. e1669 stvr.1669.
- [121] Stevens, W. P., Myers, G. J. y Constantine, L. L.: *Structured design*. IBM Systems Journal, 13(2):115–139, 1974, ISSN 0018-8670.
- [122] Strüber, D.: *Generating efficient mutation operators for search-based model-driven engineering*. En Guerra, E. y Brand, M. van den (eds.): *Theory and Practice of Model Transformation*, págs. 121–137, Cham, 2017. Springer International Publishing, ISBN 978-3-319-61473-1.
- [123] Topçu, O., Durak, U., Oğuztüzin, H. y Yilmaz, L.: *Distributed Simulation - A Model Driven Engineering Approach*. Feb. 2016, ISBN 978-3-319-03050-0.
- [124] Trætteberg, H. y Aalberg, T.: *JExercise: A specification-based and test-driven exercise support plugin for Eclipse*. En *OOPSLA Workshop on Eclipse Technology eXchange, eclipse '06*, págs. 70–74, New York, NY, USA, 2006. ACM, ISBN 1-59593-621-1.

- [125] Trakhtenbrot, M.: *New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts Models*. En *Workshop on Mutation Analysis (Mutation)*, págs. 151–160, 2007.
- [126] Troya, J., Bergmayr, A., Burgueño, L. y Wimmer, M.: *Towards systematic mutations for and with ATL model transformations*. En *International conference on software testing, verification and validation workshops (ICSTW)*, págs. 1–10, 2015.
- [127] Tuya, J., Cabal, M. J. S. y Riva, C. de la: *SQLMutation: A tool to generate mutants of SQL database queries*. En *Workshop on Mutation Analysis (Mutation)*, pág. 1, 2006.
- [128] Tuya, J., Cabal, M. J. S. y Riva, C. de la: *Mutating database queries*. *Information & Software Technology*, 49(4):398–417, 2007.
- [129] Verhoeff, T.: *Programming task packages: Peach exchange format*. Olympiads in Informatics, 2008.
- [130] Vincenzi, A. M. R., Maldonado, J. C., Barbosa, E. F. y Delamaro, M. E.: *Unit and integration testing strategies for C programs using mutation-based criteria*, págs. 45–45. Springer US, Boston, MA, 2001, ISBN 978-1-4757-5939-6.
- [131] Vincenzi, A. M. R., Simão, A. S., Delamaro, M. E. y Maldonado, J. C.: *Muta-Pro: Towards the definition of a mutation testing process*. *Journal of the Brazilian Computer Society*, 12(2):49–61, Jun. 2006, ISSN 1678-4804.
- [132] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E. y Wachsmuth, G.: *DSL engineering - designing, implementing and using domain-specific languages*. dslbook.org, 2013, ISBN 978-1-4812-1858-0.
- [133] Warmer, J. y Kleppe, A.: *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2ª ed., 2003, ISBN 0321179366.
- [134] Woodward, J. R. y Swan, J.: *The automatic generation of mutation operators for genetic algorithms*. En *14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO, págs. 67–74, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1178-6.
- [135] Wu, F., Nanavati, J., Harman, M., Jia, Y. y Krinke, J.: *Memory mutation testing*. *Information & Software Technology*, 81:97–111, 2017.

Anexo A

Gramática de WODEL

```
1 WODELPROGRAM ::= DEFINITION
2   with ( commands { MUTATION* } | blocks { BLOCK* } )
3   ( constraints { CONSTRAINT* } )?
4
5 DEFINITION ::=
6   generate (<num>|exhaustive) mutants in <folder> from SEEDS
7   metamodel <meta-model>
8   ( with resources from { RESOURCE } ( , { RESOURCE } )* )?
9   ( description <description> )?
10
11 CONSTRAINT ::=
12   context <EClass> <EString> ( : INVARIANT+ | :: <EString>+ )
13
14 INVARIANT ::=
15   MINIOCL::INVARIANT
16
17 BLOCK ::= <name> '=' <description>?
18   ( from BLOCK ( , BLOCK )* )? ( repeat '=' REPEAT )?
19   { MUTATION+ }
20   ( [ (<min> ..)? <max> ] )?
21
22 REPEAT ::= yes | no
23
24 MUTATION ::=
25   ( CREATEOBJECT | CREATEREERENCE | MODIFYINFORMATION | MODIFYSOURCEREERENCE |
26     MODIFYTARGETREFERENCE | REMOVEOBJECT | REMOVEREFERENCE | RETYPEOBJECT |
27     SELECTOBJECT | SELECTSAMPLE | COMPOSITEMUTATION )
28   ( [ (<min> ..)? <max> ] )?
29
30 CLONEOBJECT ::=
31   ( <name> '=' ) ( deep )? clone OBJECTSELECTIONSTRATEGY
32   ( in OBJECTSELECTIONSTRATEGY )?
33   ( with { FEATURESET ( , FEATURESET )* } )?
34
35 CREATEOBJECT ::=
36   ( <name> '=' )? create <EClass>
37   ( in OBJECTSELECTIONSTRATEGY ( '.' <EReference> )? )?
38   ( with { FEATURESET ( , FEATURESET )* } )?
```

```

39
40 CREATEREERENCE ::=
41   ( <name> '=' )? create reference <EReference>
42   to OBJECTSELECTIONSTRATEGY in OBJECTSELECTIONSTRATEGY
43
44 MODIFYINFORMATION ::=
45   ( <name> '=' )? modify OBJECTSELECTIONSTRATEGY
46   ( with { FEATURESET ( , FEATURESET )* } )?
47
48 MODIFYSOURCEREERENCE ::=
49   modify source <EReference> ( from OBJECTSELECTIONSTRATEGY )?
50   ( to OBJECTSELECTIONSTRATEGY )?
51
52 MODIFYTARGETREFERENCE ::=
53   modify target <EReference> ( from OBJECTSELECTIONSTRATEGY )?
54   ( to OBJECTSELECTIONSTRATEGY )?
55
56 REMOVEOBJECT ::=
57   remove OBJECTSELECTIONSTRATEGY ( 'from' OBJECTSELECTIONSTRATEGY )?
58
59 REMOVEREFERENCE ::=
60   REMOVERANDOMREF | REMOVESPECIFICREF | REMOVECOMPLETEREF
61
62 REMOVERANDOMREF ::=
63   remove one reference <EReference> in <EClass>
64
65 REMOVESPECIFICREF ::=
66   remove reference <EReference> in OBJECTSELECTIONSTRATEGY
67
68 REMOVECOMPLETEREF ::=
69   remove all reference <EReference> in <EClass>
70
71 RETYPEOBJECT ::=
72   ( <name> '=' )? retype OBJECTSELECTIONSTRATEGY
73   ( in OBJECTSELECTIONSTRATEGY )?
74   as ( <EClass> | [ <EClass> ( , <EClass> )* ] )
75   ( with { FEATURESET ( , FEATURESET )* } )?
76
77 SELECTOBJECT ::=
78   <name> '=' select OBJECTSELECTIONSTRATEGY
79   ( in OBJECTSELECTIONSTRATEGY )?
80
81 SELECTSAMPLE ::=
82   <name> '=' select sample from OBJECTSELECTIONSTRATEGY
83   ( with SAMPLECLAUSE )
84   ( { FEATURESET ( , FEATURESET )* } )?
85
86 COMPOSITEMUTATION ::= ( <name> '=' )? [ MUTATION* ]
87
88 OBJECTSELECTIONSTRATEGY ::=
89   RANDOMOBJECTSELECTION | SPECIFICOBJECTSELECTION | SPECIFICCLOSURESELECTION |
90   COMPLETEOBJECTSELECTION | TYPEDSELECTION
91
92 RANDOMOBJECTSELECTION ::=
93   one ( <EClass> | [ <EClass> ( , <EClass> )* ] )

```

```

94  ( →<EReference> ( →<EReference> )* )?
95  ( from resources )? ( where { EXPRESSION } )?
96
97  SPECIFICOBJECTSELECTION ::=
98    OBJECTEMITTER ( →<EReference> ( →<EReference> )* )?
99    ( where { expression = EXPRESSION } )?
100
101  SPECIFICCLOSURESELECTION ::=
102    closure ( OBJECTEMITTER ( →<EReference> ( →<EReference> )* )? )
103    ( where { expression = EXPRESSION } )?
104
105  COMPLETEOBJECTSELECTION ::=
106    all ( <EClass> | [ <EClass> ( , <EClass> )* ] ) →<EReference> ( →<EReference> )* )?
107    ( where { expression = EXPRESSION } )?
108
109  TYPEDSELECTION ::=
110    typed ( <EClass> | [ <EClass> ( , <EClass> )* ] )
111    ( where { expression = EXPRESSION } )?
112
113  EXPRESSION ::=
114    first = EVALUATION ( BINARYOPERATOR EVALUATION )*
115
116  BINARYOPERATOR :=
117    and | or
118
119  EVALUATION ::=
120    <name> | self | container? ( ( . <EAttribute> EVALUATIONTYPE ) |
121    ( →<EReference> OBJECTSELECTIONSTRATEGY ( →<EReference> OBJECTSELECTIONSTRATEGY )* ) )
122
123  EVALUATIONTYPE ::=
124    NUMBERTYPE | BOOLEANATYPE | STRINGTYPE | RANDOMTYPE | LISTTYPE
125
126  NUMBERTYPE ::=
127    INTEGERTYPE | DOUBLETTYPE | MINVALUETYPE | MAXVALUETYPE | RANDOMNUMBERTYPE
128
129  INTEGERTYPE ::=
130    OPERATOR <EInt>
131
132  DOUBLETTYPE ::=
133    OPERATOR <EDouble>
134
135  MINVALUETYPE ::=
136    OPERATOR min ( <EAttribute> )
137
138  MAXVALUETYPE ::=
139    OPERATOR max ( <EAttribute> )
140
141  RANDOMNUMBERTYPE ::=
142    RANDOMINTEGERTYPE | RANDOMDOUBLETTYPE
143
144  RANDOMINTEGERTYPE ::=
145    OPERATOR random-int ( <EInt> , <EInt> )
146
147  RANDOMDOUBLETTYPE ::=
148    OPERATOR random-double ( <EDouble> , <EDouble> )

```

```

149
150 BOOLEANTYPE ::=
151   SPECIFICBOOLEANTYPE | RANDOMBOOLEANTYPE
152
153 SPECIFICBOOLEANTYPE ::=
154   OPERATOR <EBoolean>
155
156 RANDOMBOOLEANTYPE ::=
157   random—boolean
158
159 STRINGTYPE ::=
160   SPECIFICSTRINGTYPE | RANDOMSTRINGTYPE | UPPERSTRINGTYPE | LOWERSTRINGTYPE |
161   STARTSTRINGTYPE | ENDSTRINGTYPE | LISTSTRINGTYPE | REPLACESTRINGTYPE
162
163 RANDOMSTRINGTYPE ::=
164   OPERATOR random—string ( <EInt> , <EInt> )
165
166 SPECIFICSTRINGTYPE ::=
167   OPERATOR <EString>
168
169 UPPERSTRINGTYPE ::=
170   OPERATOR upper
171
172 LOWERSTRINGTYPE ::=
173   OPERATOR lower
174
175 STARTSTRINGTYPE ::=
176   OPERATOR catstart ( <EString> )
177
178 ENDSTRINGTYPE ::=
179   OPERATOR catend ( <EString> )
180
181 LISTSTRINGTYPE ::=
182   OPERATOR [ <EString> ( , <EString> )* ]
183
184 REPLACESTRINGTYPE ::=
185   OPERATOR replace ( <EString> , <EString> )
186
187 RANDOMTYPE ::=
188   OPERATOR random
189
190 LISTTYPE ::=
191   OPERATOR { <EObject> ( , <EObject> )* }
192
193 OPERATOR ::=
194   = | <> | in | is | not | >= | <= | > | <

```

Listado A.1 Gramática de WODEL.

Anexo B

Gramática de EDUTEST

```
1 EDUTESTPROGRAM ::= ( EDUTESTCONFIGURATION )?
2   ( MUTATORTEST )+
3
4 EDUTESTCONFIGURATION ::= navigation '=' EDUTESTNAVIGATION
5
6 EDUTESTNAVIGATION ::= free | locked
7
8 MUTATORTEST ::= ALTERNATIVERESPONSE |
9   MULTICHOICEDIAGRAM | MULTICHOICEEMENDATION
10
11 TESTCONFIGURATION ::= retry '=' ( yes | no )
12
13 MULTICHOICEEMENDATIONCONFIG ::= retry '=' ( yes | no ) ,
14   weighted '=' ( yes | no ) ,
15   penalty '=' <EDouble> ,
16   order '=' TESTORDER ,
17   mode '=' TESTMODE
18
19 TESTORDER ::= fixed | random |
20   options—ascending | options—descending
21
22 MODEORDER ::= radiobutton | checkbox
23
24 TEST ::= description for <EString> '=' <EString>
25
26 ALTERNATIVERESPONSE ::=
27   AlternativeResponse ( WODEL::BLOCK )?
28   { TESTCONFIGURATION TEST* }
29
30 MULTICHOICEDIAGRAM ::=
31   MultiChoiceDiagram ( WODEL::BLOCK )?
32   { TESTCONFIGURATION TEST* }
33
34 MULTICHOICEEMENDATION ::=
35   MultiChoiceEmendation ( WODEL::BLOCK )?
36   { MULTICHOICEEMENDATIONCONFIG TEST* }
```

Listado B.1 Gramática de EDUTEST.

Anexo C

Gramática de MODELDRAW

```
1 MODELDRAWPROGRAM ::= metamodel <EString>
2   <EClass> : MODELDRAWTYPE
3   { NODE* RELATION* CONTENT* }
4
5 MODELDRAWTYPE ::= diagram
6
7 NODE ::= <EClass> ( '(' BOOLEANATT ( , BOOLEANATT )* ')' )? ':'
8   NODETYPE ( '=' <EAttribute> )?
9   ( compartments '=' { <EReference> ( <EReference> )* } )?
10  ( shape '=' NODESHAPE )?
11  ( color '=' NODECOLOR )?
12  ( style '=' NODESTYLE )?
13
14 BOOLEANATT ::= ( not )? <EAttribute>
15
16 RELATION ::= EDGE | HIERARCHY
17
18 EDGE ::= <EClass> '(' <EReference> , <EReference> ')'
19   edge ( '=' <EAttribute> )?
20   ( label '=' <EReference> ',' )? <EAttribute>?
21   ( src_decoration '=' DECORATION )?
22   ( src_label '=' <EAttribute> )?
23   ( tar_decoration '=' DECORATION )?
24   ( tar_label '=' <EAttribute> )?
25
26 HIERARCHY ::= <EClass> ':' <EReference> ':'
27   edge ( '=' <EAttribute> )?
28   ( label '=' <EReference> ',' )? <EAttribute>?
29   ( src_decoration '=' DECORATION )?
30   ( src_label '=' <EAttribute> )?
31   ( tar_decoration '=' DECORATION )?
32   ( tar_label '=' <EAttribute> )?
33
34 CONTENT ::= <EClass> ':' ( NODEENUMERATOR+ )?
35   ( INFORMATION+ )?
36   ( { <EAttribute> } )?
37   ( text '=' <EString> )?
38
```



```
39 NODEENUMERATOR ::= <EAttribute> '[' ENUMERATOR+ ']'
40
41 ENUMERATOR ::= <EEnumLiteral> '=' <EString>
42
43 INFORMATION ::= <EReference> ( '.' <EAttribute> )?
44
45 NODETYPE ::= node | markednode
46
47 NODESHAPE ::= circle | doublecircle | record
48
49 NODECOLOR ::= gray | ...
50
51 NODESTYLE ::= italic | underline
52
53 DECORATION ::= none | triangle | diamond | odiamond | open | empty
```

Listado C.1 Gramática de MODELDRAW.

Anexo D

Gramática de MODELTEXT

```
1 MODELTEXTPROGRAM ::= metamodel <EString>
2   ( ELEMENT+ ) *
3
4 ELEMENT ::= '>' <EString>
5   ( ' ' <EReference> )? ( '(' ATTRIBUTE ')' )? ' ' ( WORD+ )?
6
7 ATTRIBUTE ::= ( not )? <EAttribute>
8
9 WORD ::= CONSTANT | VARIABLE
10
11 CONSTANT ::= <EString>
12
13 VARIABLE ::= '%' ( <EReference> ' ' )? <EAttribute>
```

Listado D.1 Gramática de MODELTEXT.

Anexo E

Gramática de MUTATEX

```
1 MUTATEXPROGRAM ::= metamodel <EString>
2   ( OPTION+ ) *
3
4 OPTION ::= '>' <EClass> ( '(' <EString> ')' )? ':'
5   TEXT '/' TEXT
6
7 TEXT ::= ( WORDS+ ) ?
8
9 WORD ::= CONSTANT | VARIABLE
10
11 CONSTANT ::= <EString>
12
13 VARIABLE ::= %object | %attName | %oldValue | %newValue | %refName |
14   %fromObject | %oldFromObject | %srcRefName | %toObject | %oldToObject |
15   %firstRefName | %firstObject | %firstFromObject | %firstToObject |
16   %secondRefName | %secondObject | %secondFromObject | %secondToObject |
17   %firstAttName | %firstValue | %secondAttName | %secondValue |
18   %fromType | %toType
```

Listado E.1 Gramática de MUTATEX.

Anexo F

Operadores de Mutación para Autómatas Finitos

La Tabla F.1 muestra una colección de operadores de mutación para Autómatas Finitos ideados por nosotros y presentados en [104] utilizando WODEL.

Operador de mutación	Código WODEL
Crear transición	create Transition with {symbol = one Symbol}
Crear estado final	s = create FinalState create Transition with {tar = s, symbol = one Symbol}
Crear estado conectado	s = create CommonState with {name = random-string (1, 4)} create Transition with {tar = s, symbol = one Symbol}
Eliminar transición	remove one Transition
Eliminar estado y transiciones adyacentes	remove one [CommonState, FinalState] remove all Transition where {src = null } remove all Transition where {tar = null }
Cambiar símbolo de transición	modify target symbol from one Transition to one Symbol
Cambiar estado final a no-final	retype one FinalState as CommonState
Cambiar estado inicial a otro diferente	s0 = retype one InitialState as CommonState retype one State where {self <> s0} as InitialState
Intercambiar dirección de transición	modify one Transition with {swap(src, tar)}
Intercambiar símbolo de dos transiciones hermanas	t = select one Transition modify one Transition where {self <> t and src = t→src} with {swap(symbol, t→symbol)}
Redireccionar transición a nuevo estado final	s = create FinalState with {name = 'f'} modify target tar from one Transition to s
Combinación de añadir una nueva transición y cambiar el estado inicial	s0 = retype one InitialState as CommonState s1 = retype one CommonState where {self <> s1} as InitialState create Transition with {src = s1, tar = s0, symbol = one Symbol}

Operador de mutación	Código WODEL
Crear transición lambda	create Transition
Crear transición con el mismo símbolo de un estado a otro diferente	t = select one Transition where {symbol <> null} create Transition with {src = t→src, symbol = t→symbol, tar = one State where {self <> t→tar}}

Tabla F.1 Operadores de mutación para Autómatas Finitos utilizando WODEL.

Anexo G

Operadores de Mutación para el Meta-modelo de Java de MODISCO

La Tabla G.1 muestra una colección de operadores de mutación extraídos de [31, 66, 97, 108] para el meta-modelo de Java de MODISCO utilizando WODEL.

Operador de mutación	Código WODEL
Reemplazo de un operador aritmético por el operador de suma	modify one InfixExpression where {operator in ['-', '*', '/', '\ %'] and leftOperand <> null and rightOperand <> null} with {operator = '+'}
Reemplazo de un operador aritmético por el operador de resta	modify one InfixExpression where {operator in ['+', '*', '/', '\ %'] and leftOperand <> null and rightOperand <> null and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral} with {operator = '-'}
Reemplazo de un operador aritmético por el operador de multiplicación	modify one InfixExpression where {operator in ['+', '-', '/', '\ %'] and leftOperand <> null and rightOperand <> null and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral} with {operator = '*'}
Reemplazo de un operador aritmético por el operador de división	modify one InfixExpression where {operator in ['+', '-', '*', '\ %'] and leftOperand <> null and rightOperand <> null and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral} with {operator = '/'}
Reemplazo de un operador aritmético por el operador de resta	modify one InfixExpression where {operator in ['+', '-', '*', '/'] and leftOperand <> null and rightOperand <> null and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral} with {operator = '\ %'}
Reemplazo de un operador posfijo '++' por el operador posfijo '--'	modify one PostfixExpression where {operator = '++' and operand <> null} with {operator = '--'}
Reemplazo de un operador posfijo '--' por el operador posfijo '++'	modify one PostfixExpression where {operator = '--' and operand <> null} with {operator = '++'}

Operador de mutación	Código WODEL
Reemplazo de un operador prefijo '++' por el operador prefijo '-'	modify one PrefixExpression where {operator = '++' and operand <> null} with {operator = '--'}
Reemplazo de un operador prefijo '-' por el operador prefijo '++'	modify one PrefixExpression where {operator = '--' and operand <> null} with {operator = '++'}
Reemplazo de un operador prefijo '+' por el operador prefijo '-'	modify one PrefixExpression where {operator = '+' and operand <> null} with {operator = '-'}
Reemplazo de un operador prefijo '-' por el operador prefijo '+'	modify one PrefixExpression where {operator = '-' and operand <> null} with {operator = '+'}
Reemplazo de un operador prefijo '+' o '-' por el operador prefijo '-'	modify one PrefixExpression where {operator in ['+', '-'] and operand <> null and operand not typed NumberLiteral} with {operator = '--'}
Reemplazo de un operador prefijo '+' o '-' por el operador prefijo '++'	modify one PrefixExpression where {operator in ['+', '-'] and operand <> null and operand not typed NumberLiteral} with {operator = '++'}
Reemplazo de un operador prefijo '++' o '--' por el operador prefijo '+'	modify one PrefixExpression where {operator in ['++', '--'] and operand <> null} with {operator = '+'}
Reemplazo de un operador relacional por el operador relacional mayor que	modify one InfixExpression where {operator in ['>=', '<', '<=', '==', '!='] and leftOperand <> null and rightOperand <> null} with {operator = '>'}
Reemplazo de un operador relacional por el operador relacional de mayor o igual que	modify one InfixExpression where {operator in ['>', '<', '<=', '==', '!='] and leftOperand <> null and rightOperand <> null} with {operator = '>='}
Reemplazo de un operador relacional por el operador relacional menor que	modify one InfixExpression where {operator in ['>', '>=', '<', '==', '!='] and leftOperand <> null and rightOperand <> null} with {operator = '<'}
Reemplazo de un operador relacional por el operador relacional menor o igual que	modify one InfixExpression where {operator in ['>', '>=', '<', '==', '!='] and leftOperand <> null and rightOperand <> null} with {operator = '<='}
Reemplazo de un operador relacional por el operador relacional igual a	modify one InfixExpression where {operator in ['>', '>=', '<', '<=', '!='] and leftOperand <> null and rightOperand <> null} with {operator = '=='}
Reemplazo de un operador relacional por el operador relacional distinto de	modify one InfixExpression where {operator in ['>', '>=', '<', '<=', '=='] and leftOperand <> null and rightOperand <> null} with {operator = '!='}
Reemplazo de un operador condicional por el operador condicional '&&'	modify one InfixExpression where {operator in [' ', '^'] and leftOperand <> null and rightOperand <> null} with {operator = '&&'}
Reemplazo de un operador condicional por el operador condicional ' '	modify one InfixExpression where {operator in [' ', '^'] and leftOperand <> null and rightOperand <> null} with {operator = ' '}

Operador de mutación	Código WODEL
Reemplazo de un operador condicional por el operador condicional '^'	modify one InfixExpression where {operator in [' ', '^'] and leftOperand <> null and rightOperand <> null} with {operator = '^'}
Reemplazo de un operador lógico por el operador lógico '&'	modify one InfixExpression where {operator in [' ', '^'] and leftOperand <> null and rightOperand <> null} with {operator = '&'}
Reemplazo de un operador lógico por el operador lógico ' '	modify one InfixExpression where {operator in ['&', '^'] and leftOperand <> null and rightOperand <> null} with {operator = ' '}
Reemplazo de un operador lógico por el operador lógico '^'	modify one InfixExpression where {operator in ['&', ' '] and leftOperand <> null and rightOperand <> null} with {operator = '^'}
Reemplazo de un operador lógico por el operador lógico '>>'	modify one InfixExpression where {operator in ['>>>', '<<'] and leftOperand <> null and rightOperand <> null} with {operator = '>>'}
Reemplazo de un operador lógico por el operador lógico '>>>'	modify one InfixExpression where {operator in ['>>', '<<'] and leftOperand <> null and rightOperand <> null} with {operator = '>>>'}
Reemplazo de un operador lógico por el operador lógico '<<'	modify one InfixExpression where {operator in ['>>', '>>>'] and leftOperand <> null and rightOperand <> null} with {operator = '<<'}
Reemplazo de un operador de asignación por el operador de asignación '+='	modify one Assignment where {operator in ['-=', '*=', '/=', '\%='] and leftHandSide <> null and rightHandSide <> null} with {operator = '+='}
Reemplazo de un operador de asignación por el operador de asignación '-='	modify one Assignment where {operator in ['+=', '*=', '/=', '\%='] and leftHandSide <> null and rightHandSide <> null} with {operator = '-='}
Reemplazo de un operador de asignación por el operador de asignación '*='	modify one Assignment where {operator in ['+=', '-=', '/=', '\%='] and leftHandSide <> null and rightHandSide <> null} with {operator = '*='}
Reemplazo de un operador de asignación por el operador de asignación '/='	modify one Assignment where {operator in ['+=', '-=', '*=', '%='] and leftHandSide <> null and rightHandSide <> null} with {operator = '/='}
Reemplazo de un operador de asignación por el operador de asignación '%='	modify one Assignment where {operator in ['+=', '-=', '*=', '/='] and leftHandSide <> null and rightHandSide <> null} with {operator = '%='}
Reemplazo de un operador de asignación por el operador de asignación '&='	modify one Assignment where {operator in [' =', '^=', '>>=', '>>>=', '<<='] and leftHandSide <> null and rightHandSide <> null} with {operator = '&='}
Reemplazo de un operador de asignación por el operador de asignación ' ='	modify one Assignment where {operator in ['&=', '^=', '>>=', '>>>=', '<<='] and leftHandSide <> null and rightHandSide <> null} with {operator = ' ='}
Reemplazo de un operador de asignación por el operador de asignación '^='	modify one Assignment where {operator in ['&=', ' ='] and leftHandSide <> null and rightHandSide <> null} with {operator = '^='}

Operador de mutación	Código WODEL
Reemplazo de un operador de asignación por el operador de asignación '>>='	modify one Assignment where {operator in ['>>=', '<<='] and leftHandSide <> null and rightHandSide <> null} with {operator = '>>='}
Reemplazo de un operador de asignación por el operador de asignación '>>>='	modify one Assignment where {operator in ['>>=', '<<='] and leftHandSide <> null and rightHandSide <> null} with {operator = '>>>='}
Reemplazo de un operador de asignación por el operador de asignación '='	modify one Assignment where {operator in ['>>=', '>>>='] and leftHandSide <> null and rightHandSide <> null} with {operator = '<<='}
Convierte un literal numérico en un número aleatorio	modify one NumberLiteral with {tokenValue = random-int-string(0, 9)}
Elimina una llamada a un método de tipo void	remove one MethodInvocation where {method→returnType→type is typed PrimitiveTypeVoid}
Elimina una llamada a un método de un tipo distinto de void	remove one MethodInvocation where {method→returnType→type not typed PrimitiveTypeVoid}
Reemplaza una llamada a un método constructor por null	a = select one Assignment where {rightHandSide is typed ClassInstanceCreation} create NullLiteral in a→rightHandSide
Elimina una asignación	remove one Assignment where {rightHandSide is typed NumberLiteral}
Propagación de argumento en una instrucción return	stmt = select one ReturnStatement where {expression is typed MethodInvocation} method = select one MethodInvocation in stmt→expression param = select one SingleVariableAccess in method→arguments modify stmt with {expression = param}
Propagación de argumento en el operando derecho de una expresión infija	exp = select one InfixExpression where {rightOperand is typed MethodInvocation} method = select one MethodInvocation in exp→rightOperand param = select one SingleVariableAccess in method→arguments modify exp with {rightOperand = param}
Propagación de argumento en el operando izquierdo de una expresión infija	exp = select one InfixExpression where {leftOperand is typed MethodInvocation} method = select one MethodInvocation in exp→leftOperand param = select one SingleVariableAccess in method→arguments modify exp with {leftOperand = param}
Propagación de argumento en una asignación	a = select one Assignment where {rightHandSide is typed MethodInvocation} method = select one MethodInvocation in a→rightHandSide param = select one SingleVariableAccess in method→arguments modify a with {rightHandSide = param}
Incremento de un literal numérico en el operando derecho de una expresión infija	exp = select one InfixExpression where {rightOperand is typed NumberLiteral} n = select one NumberLiteral in exp→rightOperand inc = deep clone n with {tokenValue = '1'} create InfixExpression in exp→rightOperand with {leftOperand = n, operator = '+', rightOperand = inc}
Incremento de un literal numérico en el operando izquierdo de una expresión infija	exp = select one InfixExpression where {leftOperand is typed NumberLiteral} n = select one NumberLiteral in exp→leftOperand inc = deep clone n1 with {tokenValue = '1'} create InfixExpression in exp0→leftOperand with {leftOperand = n, operator = '+', rightOperand = inc}

Operador de mutación	Código WODEL
Incremento de un literal numérico en una instrucción de retorno	<pre>exp = select one ReturnStatement where {expression is typed NumberLiteral} n = select one NumberLiteral in exp→expression inc = deep clone n with {tokenValue = '1'} create InfixExpression in exp→expression with {leftOperand = n, operator = '+', rightOperand = inc}</pre>
Incremento de un literal numérico en el lado derecho de una asignación	<pre>exp = select one Assignment where {rightHandSide is typed NumberLiteral} n = select one NumberLiteral in exp→rightHandSide inc = deep clone n with {tokenValue = '1'} create InfixExpression in exp→rightHandSide with {leftOperand = n, operator = '+', rightOperand = inc}</pre>
Cambia un valor booleano a true	modify one BooleanLiteral where {value = false} with {value = true}
Cambia un valor booleano a false	modify one BooleanLiteral where {value = true} with {value = false}
Cambia un literal numérico por 1	modify one NumberLiteral where {tokenValue <> '1'} with {tokenValue = '1'}
Reemplaza una instrucción de return por un literal null	<pre>rt = select one ReturnStatement create NullLiteral in rt→expression</pre>
Reemplaza una literal numérico por 0	modify one NumberLiteral where {tokenValue <> '0'} with {tokenValue = '0'}
Reemplaza una literal numérico en el operando izquierdo de una expresión infija por -1	<pre>exp = select one InfixExpression where {leftOperand is typed NumberLiteral} modify exp with {leftOperand.tokenValue = '1'} p = create PrefixExpression with {operator = '-', operand = exp→leftOperand} modify exp with {leftOperand = p}</pre>
Reemplaza una literal numérico en el operando derecho de una expresión infija por -1	<pre>exp = select one InfixExpression where {rightOperand is typed NumberLiteral} modify exp with {rightOperand.tokenValue = '1'} p = create PrefixExpression with {operator = '-', operand = exp→rightOperand} modify exp with {rightOperand = p}</pre>
Reemplaza una literal numérico en el operando derecho de una instrucción de retorno por -1	<pre>rt = select one ReturnStatement where {expression is typed NumberLiteral} modify rt with {expression.tokenValue = '1'} p = create PrefixExpression with {operator = '-', operand = rt→expression} modify rt with {expression = p}</pre>
Reemplaza una literal numérico en el lado derecho de una asignación por -1	<pre>a = select one Assignment where {rightHandSide is typed NumberLiteral} modify a with {rightHandSide.tokenValue = '1'} p = create PrefixExpression with {operator = '-', operand = a→rightHandSide} modify a with {rightHandSide = p}</pre>
Reemplaza una literal numérico por ''	modify one StringLiteral where {escapedValue <> ''} with {escapedValue = ''}
Elimina un operador condicional unario en una sentencia if	<pre>if = select one IfStatement where {expression is typed PrefixExpression} pre = select one PrefixExpression in if→expression where {operator = '!'} exp = select one Expression in pre→operand modify if with {expression = exp}</pre>
Elimina un operador condicional unario en una instrucción de retorno	<pre>rt = select one ReturnStatement where {expression is typed PrefixExpression} pre = select one PrefixExpression in rt→expression where {operator = '!'} exp = select one Expression in pre→operand modify rt with {expression = exp}</pre>

Operador de mutación	Código WODEL
Elimina un operador condicional unario en el lado derecho de una expresión infija	<pre> inf = select one InfixExpression where {rightOperand is typed PrefixExpression} pre = select one PrefixExpression in inf→rightOperand where {operator = '!'} exp = select one Expression in pre→operand modify inf with {rightOperand = exp} </pre>
Elimina un operador condicional unario en el lado izquierdo de una expresión infija	<pre> inf = select one InfixExpression where {leftOperand is typed PrefixExpression} pre = select one PrefixExpression in inf→leftOperand where {operator = '!'} exp = select one Expression in pre→operand modify inf with {leftOperand = exp} </pre>
Elimina una instrucción	<pre> remove one Statement where {self not typed VariableDeclarationStatement} </pre>
Elimina la negación de un condicional en una instrucción if	<pre> if = select one IfStatement exp = select one InfixExpression in if→expression neg = create PrefixExpression in if→expression with {operator = '!'} par = create ParenthesizedExpression in neg→operand with {expression = exp} </pre>
Elimina la negación de un condicional en una instrucción de retorno	<pre> rt = select one ReturnStatement exp = select one InfixExpression in rt→expression neg = create PrefixExpression in rt→expression with {operator = '!'} par = create ParenthesizedExpression in neg→operand with {expression = exp} </pre>
Elimina la negación de un condicional en el operando izquierdo de una expresión infija	<pre> exp0 = select one InfixExpression exp1 = select one InfixExpression in exp0→leftOperand neg = create PrefixExpression in exp0→leftOperand with {operator = '!'} par = create ParenthesizedExpression in neg→operand with {expression = exp1} </pre>
Elimina la negación de un condicional en el operando derecho de una expresión infija	<pre> exp0 = select one InfixExpression exp1 = select one InfixExpression in exp0→rightOperand neg = create PrefixExpression in exp0→rightOperand with {operator = '!'} par = create ParenthesizedExpression in neg→operand with {expression = exp1} </pre>
Elimina una instrucción condicional	<pre> main = select one Block where {statements is typed IfStatement} if = select one IfStatement in main→statements b = select one Block in if→thenStatement modify main with {statements += b2} remove if </pre>
Resta 1 a un literal numérico en el operando izquierdo de una expresión infija	<pre> exp = select one InfixExpression where {leftOperand is typed NumberLiteral} n = select one NumberLiteral in exp→leftOperand dec = deep clone n with {tokenValue = '1'} create InfixExpression in exp→leftOperand with {leftOperand = n, operator = '-', rightOperand = dec} </pre>
Resta 1 a un literal numérico en el operando derecho de una expresión infija	<pre> exp = select one InfixExpression where {rightOperand is typed NumberLiteral} n = select one NumberLiteral in exp→rightOperand dec = deep clone n with {tokenValue = '1'} create InfixExpression in exp→rightOperand with {leftOperand = n, operator = '-', rightOperand = dec} </pre>
Resta 1 a un literal numérico en una instrucción de retorno	<pre> exp = select one ReturnStatement where {expression is typed NumberLiteral} n = select one NumberLiteral in exp→expression dec = deep clone n with {tokenValue = '1'} create InfixExpression in exp→expression with {leftOperand = n, operator = '-', rightOperand = dec} </pre>

Operador de mutación	Código WODEL
Resta 1 al lado derecho de una asignación	<pre> exp = select one Assignment where {rightHandSide is typed NumberLiteral} n = select one NumberLiteral in exp→rightHandSide dec = deep clone n with {tokenValue = '1'} create InfixExpression in exp→rightHandSide with {leftOperand = n, operator = '-', rightOperand = dec} </pre>

Tabla G.1 Operadores de mutación para el meta-modelo de Java de MoDisco utilizando WODEL.

Anexo H

Operadores de Mutación para ATL

La Tabla H.1 muestra la colección de los operadores de mutación para ATL presentados en [126] utilizando WODEL.

Operador de mutación	Código WODEL
Creación de regla	deep clone one MatchedRule with {name = random-string (4, 6)}
Eliminación de regla	remove one MatchedRule
Cambio de nombre de regla	modify one MatchedRule with {name = random-string (4, 6)}
Borrado de in pattern	remove one InPatternElement
Cambio de tipo de in pattern	sipe = select one SimpleInPatternElement type = select one OclModelElement in sipe→type cl = select one EClass from input resources where {name <> type.name} modify type with {name = cl.name}
Cambio de nombre de in pattern	modify one InPatternElement with {varName = random-string (4, 6)}
Creación de filtro de regla	p = select one InPattern where {filter is typed OperatorCallExp} opce = select one OperatorCallExp in p→filter feat = select one OclFeature sipe = select one SimpleInPatternElement in p→elements conj = create OperatorCallExp in p→filter with {operationName = 'and'} call = create OperationCallExp in conj→^source with {operationName = feat.name} exp = create VariableExp in call→^source modify sipe with {variableExp += exp} modify conj with {arguments += opce} modify p with {filter += conj}
Borrado de filtro de regla	remove one OclExpression where {container is typed InPattern}
Cambio de condición de filtro de regla	p = select all InPattern where {filter <> null} opce = select one OperationCallExp in p→filter where {operationName <> ['not', 'and', 'or']} feat = select one OclFeature where {name <> opce.operationName} modify opce with {operationName = feat.name}

Operador de mutación	Código WODEL
Creación de out pattern	<pre> cl = select one EClass from output resources p = select one OutPattern mod = select one OclModel in p→elements→type→model ope = create SimpleOutPatternElement in p→elements with {varName = random-string(2, 4)} elem = create OclModelElement in ope→type with {name = cl.name, variableDeclaration = ope} modify mod with {elements += elem} </pre>
Borrado de out pattern	remove one OutPatternElement
Cambio de tipo de out pattern	<pre> p = select one OutPattern type = select one OclModelElement in p→elements→type cl = select one EClass from output resources where {name <> type.name} modify type with {name = cl.name} </pre>
Cambio de nombre de out pattern	modify one OutPatternElement with {varName = random-string(4, 6)}
Creación de <i>binding</i>	<pre> sope = select one SimpleOutPatternElement type = select one OclModelElement in sope→type cl = select one EClass from output resources where {name = type.name} att = select one EAttribute in cl→eAllAttributes b = create Binding in sope→bindings with {isAssignment = false, propertyName = att.name} create StringExp in b→value with {stringSymbol = random-string(4, 6)} </pre>
Borrado de <i>binding</i>	remove one Binding
Cambio de valor de <i>binding</i>	<pre> b = select one Binding where {value is typed OperatorCallExp} create StringExp in b→value with {stringSymbol = random-string(4, 6)} </pre>
Cambio de propiedad de <i>binding</i>	<pre> sope = select one SimpleOutPatternElement type = select one OclModelElement in sope→type cl = select one EClass from output resources where {name = type.name} b = select one Binding in sope→bindings att = select one EAttribute in cl→eAllAttributes where {name <> b.propertyName} modify b with {propertyName = att.name} </pre>

Tabla H.1 Operadores de mutación para ATL utilizando WODEL.